



(19)

Europäisches Patentamt

European Patent Office

Office européen des brevets



(11)

EP 0 903 694 A1

(12)

EUROPEAN PATENT APPLICATION

(43) Date of publication:
24.03.1999 Bulletin 1999/12(51) Int. Cl.⁶: G06T 15/00

(21) Application number: 98107592.2

(22) Date of filing: 27.04.1998

(84) Designated Contracting States:
 AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU
 MC NL PT SE
 Designated Extension States:
 AL LT LV MK RO SI

(30) Priority: 01.08.1997 US 905238

(71) Applicant:
 MITSUBISHI DENKI KABUSHIKI KAISHA
 Tokyo 100-8310 (JP)

(72) Inventors:
 • Lauer, Hugh, C.
 Concord, MA 01742 (US)
 • Osborne, Randy B.
 Newton, Massachusetts 02159 (US)
 • Pfister, Hanspeter
 Somerville, MA 02143 (US)

(74) Representative:
 Pfenning, Meinig & Partner
 Mozartstrasse 17
 80336 München (DE)

(54) Real-time PC based volume rendering system

(57) Apparatus is provided to enable real-time volume rendering on a personal computer or a desktop computer in which a technique involving blocking of voxel data organizes the data so that all voxels within a block are stored at consecutive memory addresses within a single memory model, making possible fetching an entire block or data in a burst rather than one voxel at a time. This permits utilization of DRAM memory modules which provide high capacity and low cost with substantial space savings. Additional techniques including sectioning reduces the amount of intermediate storage in a processing pipeline to an acceptable level for semiconductor implementation. A multiplexing technique takes advantage of blocking to reduce the amount of data needed to be transmitted per block, thus reducing the number of pins and the rates at which data must be transmitted across the pins connecting adjacent processing modules with each other. A mini-blocking technique saves the time needed to process sections by avoiding reading entire blocks for voxels near the boundary between a section and previously processed sections.

Description

[0001] The copy of the basic US-application SN 08/905,238 is enclosed. With respect to the disclosure of the invention it is referred to the basic US-application.

5

FIELD OF INVENTION

[0002] This invention relates to volume graphics and more particularly to a method and apparatus for processing voxel-based data in real time and for creating visual images of objects represented by the data.

10

BACKGROUND OF THE INVENTION

[0003] Volume graphics is the subfield of computer graphics that deals with the visualization of objects or phenomena represented as sampled data in three or more dimensions. These samples are called volume elements, or "voxels," and contain digital information representing physical characteristics of the objects or phenomena being studied. For example, voxel data for a particular object may represent density, type of material, temperature, velocity, or some other property at discrete points in space throughout the interior and in the vicinity of the object.

[0004] Voxel-based representations of objects occur in many situations and applications. For example, tomographic scans and nuclear magnetic resonance scans of a human body or industrial assembly produce three dimensional arrays of data representing the density and type of the material comprising the body or object. Likewise, seismic data collected from earthquakes and controlled explosions is processed into three dimensional arrays of data representing the types of soil and rock beneath the surface of the earth. In pre-natal health care, ultrasound scans of a human fetus in the womb produce 3-D sampled data for non-invasive examination and diagnostic purposes. Still another example is the modeling of the flow of air over an aircraft wing or through a jet engine, which also results in discrete samples of data at points in three dimensional space that can be used for design and analysis of the aircraft or engine.

[0005] It is natural to want to see images of objects represented by voxels. In the past, two methods have been available for this purpose. One method is to construct a series of parallel two-dimensional image slices, each representing a slightly different cross section of the object being viewed. This is the method typically used by radiologists when viewing computed tomography scans or nuclear magnetic resonance scans of the human body. Radiologists are trained to construct three-dimensional mental pictures of the internal organs of the body from these series of two-dimensional images. The slices are, in general, parallel to one of the primary dimensions or axes of the body, so that they represent the "sagittal," "axial," and "coronal" views that are familiar to radiologists. This method of visualizing voxel-based data is difficult, requires years of training, and is prone to uncertainty, even by the most expert practitioners.

[0006] Another method is to convert voxel data into representations suitable for computer graphics system to display. Most computer graphic systems today are designed to display surfaces of objects by subdividing those surfaces into small triangles or polygons. These triangles are assigned colors and levels of transparency or opacity, then converted into pixels, that is picture elements, and projected onto the computer screen. Triangles corresponding to surfaces in the foreground obscure those corresponding to surfaces in the background. Triangles can also be colored or painted with textures and other patterns to make them look more realistic. Additional realism is made possible by simulating the position and effects of lights, so that highlights and shadows appear on the resulting image. The art and science of this kind of graphics system is well-developed and described by a large body of literature such as the textbook "Computer Graphics: Principles and Practice," 2nd edition, by J. Foley, A. vanDam, S. Feiner, and J. Hughes, published by Addison-Wesley of Reading, Massachusetts, in 1990.

[0007] This kind of polygon-based graphics system is especially suitable for displaying images of objects that are represented as computer models of their surfaces, such as architectural or mechanical drawings. However, it is less appropriate for visualizing objects represented by 3-D sampled data or voxels, because the process of converting the sample to triangles or polygons is itself computationally expensive. Many algorithms exist for performing the conversion from voxels to polygons, including the famous Marching Cubes algorithm described by W. E. Lorensen and H. E. Cline in a paper entitled "Marching Cubes: A high-resolution 3D surface construction algorithm," presented in Computer Graphics, the Proceedings of the 1987 SIGGRAPH Conference, pages 163 - 169. All of these algorithms suffer the problem of losing detail of the surface, something that would be intolerable in applications such as medical imaging and others.

[0008] In recent years, an alternative method has emerged called volume rendering. This method is a form of digital signal processing in which the individual voxels of a voxel-based representation are assigned colors and levels of transparency or opacity. They are then projected on a two-dimensional viewing surface such as a computer screen, with opaque voxels in the foreground obscuring other voxels in the background. This accumulation of projected voxels results in a visual image of the object. Lighting calculations can be done on the individual voxels to create the appearance of highlights and shadows in a similar manner to that of conventional computer graphics.

[0009] By changing the assignment of colors and transparency to particular voxel data values, different views of the

exterior and interior of an object can be seen. For example, a surgeon needing to examine the ligaments, tendons, and bones of a human knee in preparation for surgery can utilize a tomographic scan of the knee and cause voxel data values corresponding to blood, skin, and muscle to appear to be completely transparent. In another example, a mechanic using a tomographic scan of a turbine blade or weld in a jet engine can cause voxel data values representing solid metal to appear to be transparent while causing those representing air to be opaque. This allows the viewing of internal flaws in the metal that would otherwise be hidden from the human eye.

[0010] The process of creating a viewable image from computer data is called "rendering," and the process of creating a viewable image from voxel data is called "volume rendering." The mechanism for mapping the data values of individual voxels to colors and transparencies is called a "transfer function."

10

a) Projection of Voxel Data

[0011] There are a number of techniques to take the data points or voxels representing an object and project them onto a flat viewing surface such as a computer screen. In each of these techniques, an object to be viewed is positioned relative to the viewing surface by translating the three dimensional sampled data representing that object to the spatial coordinates of the space in front of or behind the viewing surface. The techniques are different method of computing the color and intensity of the light at discrete points or "pixels" on that viewing surface.

[0012] One technique is to compute a series of fast Fourier transforms of the voxel data, combine them, then compute the inverse Fourier transform to obtain the resulting two-dimensional image. This is described by T. Malzbender in US Patent #5,414,803 entitled "Method Utilizing Frequency Domain Representation for Generating Two-Dimensional Views of Three-Dimensional Objects."

[0013] A second technique called "splatting" was described by L. A. Westover in a Doctoral Dissertation entitled "Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm" presented to and published by the Department of Computer Science of the University of North Carolina in July 1991, Technical Report number TR91-029. In the splatting technique, each individual voxel of a set of three-dimensional sampled data is projected in the direction of the eye of the viewer. The colors and transparency of the projected voxel are mathematically combined with the pixels of the viewing surface in the immediate region surrounding the point where that projection intersects that computer screen. When all voxels are thus accumulated, the resulting image appears to be a two-dimensional picture of a three-dimensional object.

[0014] A third technique is to convert the three-dimensional set of data into a so-called "texture map" and then to store it in the texture map memory that can be found in certain types of modern computer systems. Then this texture map is used to "paint" or "color" a series of parallel planes, each perpendicular to the viewing direction, so that each appears to be a cross-section of the object in question. These planes are then mathematically combined by the graphics subsystem of the computer system to form an image of what appears to the viewer to be a three dimensional object. This method is described in detail in a paper entitled "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," presented by B. Cabral, N. Cam, and J. Foran at the "Workshop on Volume Visualization" in 1991. It is further described by T. J. Cullip and U. Neumann in a technical report number TR93-027 entitled "Accelerating volume reconstruction with 3D texture mapping hardware," published by the Department of Computer Science of the University of North Carolina at Chapel Hill.

[0015] A fourth technique is called "ray-casting." In this technique, imaginary rays are passed from the eye of the viewer through the exact center of each pixel of the viewing surface, then through the object to be viewed. Each ray which passes through the volume is "loaded up" with the visual characteristics of each point along its path. As the ray passes-through the volume, its total characteristic is the sum or mathematical integral of the characteristics of all of the points along the ray. This sum or integral is then assigned to the pixel through which the ray passes, causing a point of light to appear on the viewing surface. The accumulation of all such rays produces a visible image on the viewing surface.

[0016] When rays come through a volume, some pass between points represented by the three dimensional sampled data, not intersecting them exactly. It will be appreciated that these "missed" data points or voxels are not reflected in the color or intensity of the pixel corresponding to any ray. In order to solve this missed data-point problem, interpolation techniques are utilized to synthetically generate values from voxels in the immediate neighbourhoods of the missed points. In one example, a synthetic value is generated for each plane of sample points or voxels crossed by the ray by the mathematical method of bilinear interpolation of the values of the four nearest voxels in that plane. In another example, synthetic points are generated with uniform spacing along the ray by the mathematical method of trilinear interpolation of the eight nearest voxels surrounding each point. In these ways, as the ray passes through the object, the characteristics accumulated along the way take into account characteristics of the nearest neighbors to synthetically generate a value for the missed point. It will be appreciated that there are many possible ways of generating synthetic points and that these have a significant bearing on the quality and realism of the projected image.

[0017] In order for a two-dimensional picture to-be perceived by the human eye as the image of a three-dimensional

object or scene, it is important for the picture to include the effects of lighting and shadows. This is the subject of extensive literature in computer graphics, including the aforementioned textbook by J. Foley, et. al. Most techniques revolve around the notion of finding the "normal vector" or perpendicular direction to each point on each surface of the object being displayed, then making calculations based on these normal vectors and on the positions of the viewer and the light sources in order to illuminate those points, creating the effect of highlights and shadows.

[0018] Whereas in conventional computer graphics based on polygons and surface, these normal vectors can be calculated directly from the mathematical models of the surfaces, in volume graphics the normal vectors must be extracted from the sampled data itself. This must be done for each voxel, for example, by examining the values of the other voxels in its immediate neighborhood. At the boundaries of different materials, for instance different tissues, there will be significant differences or gradients in the values of the neighboring voxels. From these differences, the normal vectors can be calculated. Then whenever one type of material is transparent while an adjacent material is opaque, the projection can make clear the edges and surfaces between the different materials. Moreover, the lighting calculations based on these normal vectors can emphasize the irregularities of these surfaces in such a way as to be recognizable by the human eye as three dimensional. For instance, ridges in the grey matter making up the brain can be clearly displayed in this manner from a tomographic scan by simply making the skin and bone of the skull transparent.

b) Computational Requirements

[0019] It will be appreciated that all four of the above techniques for projecting voxel data onto a computing surface require massive amounts of computation and have been heretofore unsuitable for equipment of the size and cost of personal or desktop computers. Moreover, they involve the invocation of many different techniques in order to render the volume in a manner useful, for instance, in medical diagnosis. In general, each voxel of a three dimensional data set must be examined at least once to form the projected image. If the sampled data set were a cube with 256 data points on a side, this being a typical size for current tomographic and nuclear magnetic resonance scans for medical purposes, then a total of 256^3 or approximately 16 million voxels must be evaluated. If, however, the sampled data set were a cube with 4096 data points on a side, this being typical of geological data used in exploration for oil and gas, then a total of 4096^3 or approximately 64 billion voxels must be evaluated, just to render a single image.

[0020] It will be further appreciated that if rendering static images of static data is computationally expensive, this pales into insignificance when considering the computational power required to render objects that move, rotate, or change in some other way. Many applications need visualization of objects that appear to move in real time, which means rendering on the order of 30 frames per second. That is, each voxel must be re-evaluated or projected 30 times per second. For a volume of 256^3 data points, this means that data must be retrieved from the sampled data set $256^3 \times 30$ or approximately 503 million times per second. Noting that if the volume rendering were done by a computer program, between 10 and 100 computer instructions would be required per data point per frame. Therefore, the processing power to view rotating or changing volume graphics images is between five and fifty billion operations per second. Note for each doubling of the number of data points on the side of a cubic data set, the required processing power goes up by a factor of eight.

[0021] The usual compromise is to sacrifice frame rate or visual quality or cost and size. Presently, the best that one can obtain by rendering a 256^3 volume in computer software is one to two frames per second on eight ganged processors of the type found in current high-end personal computers. With very expensive computers particularly specialized for graphics and containing very large amounts of texture memory, frame rates of up to fifteen frames per second can be achieved by sacrificing lighting and shadows. Other approaches that actually achieve real-time frame rates of 30 frames per second or more without sacrificing image quality have resulted in very specialized systems that are too large and costly for personal or desktop-size equipment.

c) Reduction in Computational Requirements

[0022] In order to improve upon this rather dismal prospect for obtaining real-time volume-rendering at 30 frames per second based on the ray-casting technique, a development by Ari Kaufmann and Hanspeter Pfister at State University of New York is described in U.S. Patent #5,594,842, "Apparatus and Method for Real-time Volume Visualization." In this development, improvements can be obtained by passing a large number of rays through a volume in parallel and processing them by evaluation the volume data a slice at a time. If one can do slice-processing fast in specialized electronic hardware, as opposed to software, it has been demonstrated that one can increase from two frames per second to 30 frames per second at a modest cost.

[0023] In theory, this is accomplished in hardware through the utilization of a multiplicity of memory modules and specialized processing pipelines. Utilizing large numbers of memory modules and pipelines, one can pick out data in parallel from different memory modules in a system now dubbed "Cube-4" which was described by H. Pfister, A. Kaufmann, and T. Wessels in a paper entitled "Towards a Scalable Architecture for Real-time Volume Rendering" presented at the

10th Eurographics Workshop on Graphics Hardware at Maastricht, The Netherlands, on August 28 and 29, 1995, and further described in a Doctoral Dissertation submitted by Hanspeter Pfister to the Department of Computer Science at the State University of New York at Stony Brook in December 1996.

5 [0024] The essence of the Cube-4 system is that the three dimensional sampled data representing the object is distributed across the memory modules by a technique called "skewing," so that adjacent voxels in each dimension are stored in adjacent memory modules. Each memory module is associated with its own processing pipeline. Moreover, voxels are organized in the memory modules so that if there are a total of P pipelines and P memory modules, then P adjacent voxels can be fetched simultaneously, in parallel, within a single cycle of a computer memory system, independent of the viewing direction. This reduces the total time to fetch voxels from memory by a factor of P. For example, 10 if the data set has 256^3 voxels and P has the value four, then only $256^3 \div 4$ or approximately four million memory cycles are needed to fetch the data in order to render an image.

15 [0025] An additional characteristic of the Cube-4 system is that the computational processing required for volume rendering is organized into pipelines with specialized functions for this purpose. Each pipeline is capable of starting the processing of a new voxel in each cycle. Thus, in the first cycle, the pipeline fetches a voxel from its associated memory module and performs the first step of processing. Then in the second cycle, it performs the second step of processing of this first voxel, while at the same time fetching the second voxel and performing the first step of processing this voxel. Likewise, in the third cycle, the pipeline performs the third processing step of the first voxel, the second processing step of the second voxel, and the first processing step of the third voxel. In this manner, voxels from each memory module progress through its corresponding pipeline in lock-step fashion, one after the another, until they are fully processed. 20 Thus, instead of requiring 10 to 100 computer instructions per voxel, a new voxel can be processed in every cycle.

25 [0026] A further innovative characteristic of the Cube-4 system is that each pipeline communicates only with its nearest neighbors. Such communication is required, for example, to transmit voxel values from one pipeline to the next for purposes of estimating gradients or normal vectors so that lighting and shadow effects can be calculated. It is also used to communicate the values of rays as they pass through the volume accumulating visual characteristics of the voxels in the vicinities of the areas through which they pass.

30 [0027] This approach of nearest neighbor communication provides the Cube-4 one its principal advantages, that of being "scalable." That is, in order to accommodate larger amounts of three dimensional sampled data and/or in order to process this data faster, it is only necessary to add more memory modules and pipelines. There are no common busses or other system resources to be overloaded by the expansion.

35 [0028] In the Cube-4 system, volume rendering proceeds as follows. Data is organized as a cube or other rectangular solid. Considering first the face of this cube or solid that is most nearly perpendicular to the viewing direction, a partial row of P voxels at the top corner is fetched from P memory modules concurrently, in one memory cycle, and inserted into the first stage of the P processing pipelines. In the second cycle these voxels are moved to the second stage of their pipelines and/or transmitted to the second stages of adjacent pipelines. At the same time, the next P voxels are fetched from the same row and inserted into the first stage of their pipelines. In each subsequent cycle, P more voxels are fetched from the top row and inserted into their pipelines, while previously fetched voxels move to later stages of their pipelines. This continues until the entire row of voxels has been fetched. Then the next row is fetched, P voxels at a time, then the next and so on, until all of the rows of the face of the volume data set have been fetched and inserted into their processing pipelines.

40 [0029] This face is called a "slice." Then the Cube-4 system moves again to the top corner, but this time starts fetching the P voxels in the top row immediately behind the face, that is from the second "slice." In this way, it progresses through the second slice of the data set, a row at a time and within each row, P voxels at time. After completing the second slice, it proceeds to the third slice, then to subsequent slices in a similar manner, until all slices have been processed. The purpose of this approach is to fetch and process all of the voxels in an orderly way, P voxels at a time, until the entire 45 volume data set has been processed and an image has been formed.

[0030] In the terminology of the Cube-4 system, a row of voxels is called a "beam" and a group of P voxels within a beam is called a "partial beam."

[0031] The processing stages of the Cube-4 system perform all of the calculations required for the ray-casting technique, including interpolation of samples, estimation of the gradients or normal vectors, assignments of colors and 50 transparency or opacity, and calculation of lighting and shadow effects to produce the final image on the two dimensional view surface.

[0032] The Cube-4 system was designed to be capable of being implemented in semiconductor technology. However, two limiting factors prevent it from achieving the small size and low cost necessary for personal or desktop-size computers, namely the rate of accessing voxel values from memory modules and the amount of internal storage required in 55 each processing pipeline. With regard to the rate of accessing memory, current semi-conductor memory devices suitable for storing a volume data set in a Cube-4 system are either too slow or too expensive or both. Much cheaper memory solutions are needed for a practical system usable in a personal or desktop computer. With regard to the internal storage, the Cube-4 algorithm requires that each processing pipeline store intermediate results within itself during

processing, the amount of storage being proportional to the area of the face of the volume data set being rendered. For a 256^3 data set, this amount turns out to be so large that it would increase the size of a single-chip processing pipeline by an excessive amount and therefore to an excessive cost for a personal computer system. A practical system requires a solution for reducing this amount of intermediate storage.

5

d) Blocking and SRAM Technology

[0033] In other experimental systems designed at about the same time as Cube-4, these limitations have been ignored. One such system is called "DIV²A," the Distributed Volume Visualization Architecture, and was described in a paper by J. Lichtermann entitled "Design of a Fast Voxel Processor for Parallel Volume Visualization" presented at the 10th Eurographics Workshop on Graphics Hardware, August 28 and 29, 1995, at Maastricht, The Netherlands. Another such system is the VIRIM system, described by M. deBoer, A. Gröpl, J. Hesser, and R. Männer in a paper entitled "Latency- and Hazard-Free Volume Memory Architecture for Direct Volume Rendering," presented at the 11th Eurographics Workshop on Graphics Hardware on August 26-27, 1996, in Poitiers, France.

15

[0034] The DIV²A system comprises sixteen processing pipelines connected together in a ring, so that each pipeline can communicate directly with its nearest neighbor on each side. Each processing pipeline has an associated memory module for storing a portion of the volume data set. Voxels are organized into small subcubes, and these subcubes are distributed among the memory modules so that adjacent subcubes are stored in adjacent memory modules in each of the three dimensions. However, in order to achieve the required memory access rate for rendering a 256^3 data set at 30 frames per second, the DIV²A system requires eight parallel memory banks within each memory module. Moreover, each memory bank is implemented with as Static Random Access Memory or SRAM device.

20

[0035] In current semiconductor technology, SRAM devices are very fast, so they can support high rates of data access, but they are also very expensive, very power-hungry, and have limited capacity. Since the DIV²A system requires eight of these per processing pipeline and has sixteen processing pipelines, a total of 128 SRAM devices are needed, just to store the voxels of a 256^3 volume data set. It will be appreciated that this far exceeds the physical size and power limitations of a board that could be plugged into the back of a personal computer. Systems such as DIV²A and VIRIM are the size of a drawer of a file cabinet, not including the desktop computer to which they are connected.

SUMMARY OF THE INVENTION

30

[0036] In order to make real-time volume rendering practical for personal and desktop computers, the subject invention further improves upon the Cube-4 system by providing techniques including architecture modification to permit the use of high capacity, low cost Dynamic Random Access Memory or DRAM devices for memory modules. DRAM devices or "chips" are capable of storing five to ten times more information per chip than SRAM devices, cost five to ten times less per bit of information stored, and required considerably less power to operate. DRAM devices are currently available with capacities of 4, 16, and 64 megabits. Utilizing four 64-megabit DRAMs, only four chips are needed to store a data set of 256^3 voxels with sixteen bits per voxel. By coupling four DRAM modules with four custom designed semiconductor devices for processing pipelines, the subject invention makes it possible to implement a real-time volume rendering system on a board that can be plugged into the back of personal computer.

35

[0037] However, DRAM devices or chips are also much slower than SRAM devices. Normally, a DRAM chip can support only eight to twelve million accesses per second, versus 50 to 200 million per second for an SRAM chip. Therefore, although four DRAM devices have enough capacity to store a volume data set of 256^3 voxels, together they can support only about 32 to 48 million accesses per second, far fewer than the data rate of 503 million accesses per second needed to render the data set at 30 frames per second.

40

[0038] In order to achieve the benefits of the high capacity and low cost of DRAMs, the subject invention utilizes DRAM chips that support "burst mode" access. This feature is now found in some DRAM products and enables access rates as fast as those of SRAM devices, but only when accessing consecutive memory locations in rapid sequence. In order to be able to satisfy this condition and therefore to be able to take advantage of burst mode DRAMs, other architectural modifications of the Cube-4 system are required. The subject invention utilizes four techniques for this purpose.

45

[0039] In the first technique, called "blocking," voxel data is organized into blocks so that all voxels within a block are stored at consecutive memory addresses within a single memory module. This makes it possible to fetch an entire block of data in a burst rather than one voxel at a time. In this way, a single processing pipeline can access memory at data rates of 125 million or more voxels per second, thus making it possible for four processing pipelines and four DRAM modules to render 256^3 data sets at 30 frames per second.

50

[0040] A second technique to improve upon the Cube-4 system is called "sectioning." In this technique, the volume data set is subdivided into sections and rendered a section at a time. Because each section presents a face with a smaller area to the rendering pipeline, less internal storage is required. Moreover, intermediate results from processing individual sections can be stored outside the processing pipeline and later combined with each other to form a complete

image of the object being rendered. The effect if this technique is to reduce the amount of intermediate storage in a processing pipeline to an acceptable level for semiconductor implementation.

[0041] The third technique reduces the number of pins and the rates at which data must be transmitted across the pins connecting adjacent processing pipelines with each other. This is done by taking advantage of a side-effect of blocking that reduces the amount of data needed to be transmitted per block by a factor of approximately $1/B$, where B is the number of voxels on the edge of any block.

[0042] A final technique called "mini-blocking" is utilized which further refines the aforementioned sectioning technique. In this technique, the B^3 voxels of a block are further organized into small blocks or cubes called mini-blocks of size $2 \times 2 \times 2$ each. This makes it possible, when processing the voxels near the boundary between a section and previously processed sections, to avoid reading entire blocks but only read mini-blocks. This saves approximately five to seven percent of the time needed to process a volume in sections.

[0043] The overall effect of these four architectural improvements is to enable the implementation of a practical, low cost volume rendering system based on the Cube-4 architecture using DRAM memory modules, thereby reducing its size and cost from that of a small file cabinet to that of a printed circuit board that can be plugged into the back of a personal computer. An additional effect is to make it possible to further shrink the Cube-4 architecture so that a pipeline and its memory can be implemented within the same semiconductor chip. This reduces the size and power requirements of the volume rendering system even more.

[0044] More particularly with respect to blocking, current burst mode DRAM devices can operate at more than 125 million accesses per second, but only while fetching data from consecutive memory addresses. Minimum burst sizes are at least eight consecutive accesses but are often more. This data rate is sufficient for four pipelines to render a 256^3 data set at 30 frames per second, or approximately 503 million voxels per second. However, burst mode will only work if data can be organized in memory so that consecutive accesses are at consecutive addresses for at least a minimum burst size.

[0045] In Cube-4 as originally presented, this is not possible for most viewing directions. From at least one third of the viewing directions, consecutive voxels are accessed from memory locations N addresses apart, where N is the number of voxels on the edge of a cubic data set. From another third of the viewing directions, consecutive voxels are accessed from addresses N^2 apart. The result is that even a burst mode DRAM device is reduced to its slowest mode, that of accessing random addresses. In this mode, a DRAM device can support only about eight to twelve million accesses per second.

[0046] This problem can be solved by organizing voxel data in such a way that no matter from what direction the object is viewed, bursts of voxels can be fetched from consecutive memory addresses. To do this, voxel data is grouped into small cubes or blocks with B voxels on a side, so that all of the voxels of a block of size $B \times B \times B$ are stored at consecutive addresses within a single memory module. Although any value for B can be used, values of B equal to four or eight are most practical.

[0047] In order to preserve the Cube-4 characteristic that rendering is independent of the view direction, data must still be skewed across memory modules. However, instead of skewing by voxel as in Cube-4, data in the subject invention is skewed by block. In particular, adjacent blocks in any of the three dimensions of the volume data set are stored in adjacent memory modules. This makes it possible for P adjacent processing pipelines to fetch P adjacent blocks of voxels in burst mode from P adjacent memory modules, all concurrently and simultaneously. Each pipeline renders all of the voxels in a block, then they all step to their respective next blocks, in much the same way as the Cube-4 processing pipelines step to their respective next voxels. In this scheme, the order of processing individual voxels is not the same in the subject invention as the order in the Cube-4 system, so important modifications to the Cube-4 algorithm are necessary.

[0048] With regard to sectioning, this deals with the issue of the amount of storage needed for intermediate results in a processing pipeline. In both the Cube-4 system and the modifications introduced in the subject invention, the required amount of intermediate storage is approximately proportional to N^2/P , where N is the number of voxels on a side of a cubic data set and P is the number of memory modules and pipelines. The reason for this ratio is that in order to compute gradients or normal vectors according to the Cube-4 algorithm, it is necessary to mathematically combine the values of voxels of the slice being currently processed along with the values of voxels fetched in the two previous slices.

[0049] Similarly, in order to compute trilinear interpolations to obtain values for "missed data points" along the rays, it is necessary to mathematically combine values of voxels from the slice being processed with values from the slice previously processed. The net effect is that each pipeline must remember the voxels from one or two previously read slices in order to complete the processing of a current slice. For a cubic data set with N voxels on a side, the number of voxel values needed to be retained is proportional to the number of voxels in a slice, that is, to N^2 . However, this data is distributed across P processing pipelines, so that this number is reduced to N^2/P . The constant of proportionality is typically more than three.

[0050] More generally, if the voxel data set represents a rectangular solid of arbitrary proportions, the amount of data that must be stored from slice to slice is proportional to the area of the face most nearly perpendicular to the view direc-

tion.

[0050] It will be appreciated that for current semiconductor technology, this is much too much data to fit economically on one processing chip. In particular, if N is 256 and P is 4, then the amount of storage required in the Cube-4 system is at least $3 \times 256^2 \times 4$, or almost 50,000 voxels, equivalent to approximately 800,000 bits for 16-bit voxel values. While this amount of storage can be easily achieved with SRAM or DRAM semiconductor technology, it would result in an excessively large semiconductor device in current technology appropriate for processing units, and therefore it would be too expensive for personal and desktop computing environments.

[0051] To solve this problem, the volume data set is partitioned into sections by subdividing it perpendicular to the face of the volume nearest to the viewing direction, each such subdivision being called a "section." Sections are rendered separately from other sections, almost as if they were independent volume data sets. When rendering a section, some rays pass through the section and out the back. The visual characteristics of these rays, i.e., color, brightness, and transparency, are assigned directly to the corresponding pixels of the viewing surface. Other rays, however, pass out of a side, top, or bottom surface of the section and into an adjacent section. The visual characteristics of these rays must be saved and utilized when rendering the continuation of the same rays in the adjacent section. To save these rays, a processing pipeline of the subject invention writes them into an external storage module. Later, as it begins processing the adjacent section, it re-reads them to initialize the visual characteristics of their continuations.

[0052] The net effect is to reduce the amount of storage required within the processing pipeline to an amount proportional to the surface area of the face of the largest section. Conversely, the size of a section can be chosen based on the amount of available memory. In the subject invention, a section is approximately one quarter to one eighth of the entire volume data set.

[0053] An additional benefit of sectioning is in rendering voxel data sets that are larger than the total amount of memory in the volume memory modules. This can be done by rendering the voxel data set a section at a time. After earlier sections are rendered and their data are no longer needed, later sections are loaded into the volume memory modules, over-writing the earlier sections. These later sections are then processed, while still others sections are loaded, etc. In this way, an entire large volume data set can be passed through a smaller volume memory during the rendering process, and the resulting image can be accumulated in the external memory modules. The size of the largest volume data set that can be thus processed is limited only by the storage capacity of the in the external memory modules.

[0054] With regard to the number of pins needed to interconnect adjacent processing pipeline chips, these represent a significant component of the cost of a semiconductor device. The Cube-4 algorithm requires several hundred pins to transmit information between adjacent pipelines. These pins carry values of voxels, values computed from several voxels, and the partial accumulation of the characteristics of each ray.

[0055] In the Cube-4 algorithm, one data element must be transmitted across each set of pins for each voxel read from memory. With voxels being read in burst mode from DRAM memory at 125 megahertz, that is 125 million voxels per second, this implies a circuit with several hundred pins at operating 125 MHz between each pair of processing pipelines. This presents a serious challenge to designer of the circuit board that contains the volume rendering system.

[0056] The solution in the subject invention is to take advantage of a side effect of the blocking algorithm, namely the reduction by a factor of approximately $1/B$ of the amount of data transmitted between adjacent pipelines. This reduction occurs because data needs only to be transmitted from the voxels on the face of each block to adjacent processing pipelines. Data from voxels interior to a block are utilized only within each block. It will be appreciated that for every B^3 voxels in a block, there are only B^2 voxels on each face of that block. Therefore, the number of pieces of information that need to be transmitted to neighboring pipelines is proportional to B^2 . This results in reduction of the communication between pipeline by a factor of approximately B^2/B^3 , that is $1/B$. This factor of $1/B$ can be applied either to reducing the bandwidth of the transmitted data on individual pins or to reducing the number of pins by multiplexing. Note that any of a number of widely known multiplexing techniques may be utilized in this regard.

[0057] It will be appreciated that in order to process a section, the values of immediately adjacent voxels of previously processed sections must be utilized. These are combined mathematically with voxels values of the section being processed to obtain gradients or normal vectors in the vicinity of the edge of the section and to obtain values for "missed points" between two sections. One way to obtain these values is to re-read the voxel data of previously processed sections directly from the memory modules holding the volume data set. However, as a result of the aforementioned blocking technique, voxel values are read in bursts of a block at a time. If the value of B, the number of voxels on a side of a block, is greater than two, this causes the sectioning mechanism to re-read too many voxels, thereby wasting time and processing power.

[0058] This leads to the fourth technique for improving the Cube-4 system, namely the utilization of mini-blocks and taking advantage of the fact that in some DRAM products, the minimum burst size is eight accesses. In the subject invention, each block is subdivided into mini-blocks of size $2 \times 2 \times 2$ voxels such that each mini-block of a block is also stored in consecutive memory locations within its volume memory module. Then as the sectioning mechanism re-reads voxels from previously processed adjacent sections, it needs to re-read only the mini-blocks at the exact boundaries of those adjacent sections, not the entire blocks. It is estimated that this technique saves approximately five to seven per-

cent of the processing time of a volume data set, although the actual savings depends upon the areas of the boundaries of the sections.

[0059] It will be appreciated that an apparent simplification of the subject invention would be to set value of B, the number of voxels on an edge of a block, to two. This would appear to obviate the need for a separate mini-blocking mechanism. However, this simplification is illusory, because the savings in number of data pins between adjacent processing pipelines is determined by the factor 1/B. If B were set to two, this savings would only one-half, an amount insufficient for a low cost implementation. Thus, the economics of semiconductor design and production indicate that B should be set to a larger value, such as eight, and that a separate mini-block scheme should be implemented to avoid wasting time re-reading unnecessary voxels at the boundaries of sections.

[0060] In summary, apparatus is provided to enable real-time volume rendering on a personal computer or a desktop computer in which a technique involving blocking of voxel data organizes the data so that all voxels within a block are stored at consecutive memory addresses within a single memory model, making possible fetching an entire block of data in a burst rather than one voxel at a time. This permits utilization of DRAM or memory models which provide high capacity and low cost with substantial space savings. Additional techniques including sectioning reducing the amount of intermediate storage in a processing pipeline to an acceptable level for semiconductor implementation. A multiplexing technique takes advantage of blocking to reduce the amount of data needed to be transmitted per block, thus rendering the number of pins and the rates at which data must be transmitted across the pins connecting adjacent processing modules with each other. Mini blocking saves the time needed to process and sections by avoiding reading entire blocks for voxels near the boundary between a section and previously processed sections.

BRIEF DESCRIPTION OF THE DRAWINGS

[0061] These and other features of the Subject Invention will be better understood in connection with the Detailed Description taken in conjunction with the Drawing of which:

- 25 Figure 1 is a diagrammatic illustration of a view of a volume data set being projected onto an image plane by means of ray-casting.
- 30 Figure 2 is a diagrammatic illustration the processing of an individual ray by ray-casting.
- 35 Figure 3 is a diagrammatic illustration several methods of paralleling the processing of rays in a ray-casting system.
- 40 Figure 4 is a diagrammatic illustration the skewing of voxels among memory modules in the prior art Cube-4 system.
- 45 Figure 5 is a diagrammatic illustration slice parallel rendering as implemented in a Cube-4 system.
- 50 Figure 6 is a diagrammatic illustration the order of fetching and processing voxels from two consecutive slices in a Cube-4 system.
- 55 Figure 7 is a block diagram of a Cube-4 system, showing the connection of processing units in a ring.
- Figure 8 is a block diagram of the processing pipeline of a Cube-4 system, showing the principal components of the pipeline.
- Figure 9 is a diagrammatic illustration the memory addresses of voxels on the XY face of a volume data set in the Cube-4 system.
- 50 Figure 10 is a diagrammatic illustration the memory addresses of voxels on the YZ face of a volume data set in the Cube-4 system.
- Figure 11 is a diagrammatic illustration the organization of voxels by blocks and the skewing of blocks among memory modules in the subject invention.
- 55 Figure 12 is a block diagram of a processing pipeline and its associated memory in one embodiment of the subject invention.

EP 0 903 694 A1

Figure 13 is a diagrammatic illustration the determination of the rendering coordinates and the selected "front" face for rendering, based on the angle of a ray from the viewing surface.

5 Figure 14 is a diagrammatic illustration the base plane of a volume data set and its relation to the image plane.

Figure 15 is a diagrammatic illustration the calculation of samples from a block of B^3 voxels and voxels forwarded from previously processes blocks.

10 Figure 16 is a diagrammatic illustration the forwarding of information from a block to support the processing of adjacent blocks in the rightward, downward, and rearward directions.

Figure 17 is a diagrammatic illustration the calculation of gradients from a block of B^3 samples and samples forwarded from previously processed blocks.

15 Figure 18 is a diagrammatic illustration of three volume data sets, each subdivided into sections.

Figure 19 is a block diagram of modification to the illustration of Figure 12 showing changes needed to implement the technique of sectioning.

20 **DETAILED DESCRIPTION**

[0062] Referring now to Figure 1, a two-dimensional view of a three-dimensional volume data set 10 is shown. The third dimension of volume data set 10 is perpendicular to the printed page so that only a cross section of the data set can be seen in the figure. Voxels are illustrated by dots 12 in the figure and are data values that represent some characteristic of a three dimensional object 14 at fixed points of a rectangular grid in three dimensional space. Also illustrated in Figure 1 is a one-dimensional view of a two-dimensional image plane 16 onto which an image of object 14 is to be formed. In this illustration, the second dimension of image plane 16 is also perpendicular to the printed page.

[0063] In the technique of ray-casting, rays 18 are extended from pixels 22 the image plane 16 through the volume data set 10. Each ray accumulates color, brightness, and transparency or opacity at sample points 20 along that ray.

30 This accumulation of light determines the brightness and color of the corresponding pixels 22.

[0064] It will be appreciated that although Figure 1 suggests that the third dimension of volume data set 10 and the second dimension of image plane 16 are both perpendicular to the printed page and therefore parallel to each other, in general this is not the case. The image plane may have any orientation with respect to the volume data set, so that rays 18 may pass through volume data set 10 at any angle in all three dimensions.

35 [0065] It will also be appreciated that sample points 20 do not necessarily intersect exactly with the fixed points represented by voxels 12. Therefore, the value of each sample point must be synthesized from the values of voxels nearby. That is, the intensity of light, color, and transparency or opacity at each sample point 20 must be calculated by a mathematical function of the values of nearby voxels 12. The sample points 20 of each ray 18 are then accumulated by another mathematical function to produce the brightness and color of the pixel 22 corresponding to that ray. The resulting set of pixels 22 forms a visual image of the object 14 in the image plane 16.

[0066] In both the Cube-4 system and in the subject invention, the calculation of the color, brightness or intensity, and transparency of sample points 20 is done in two parts. First, the mathematical function of trilinear interpolation is utilized to take the weighted average of the values of the eight voxels in a cubic arrangement immediately surrounding the sample point 20. The resulting average is then used to assign a color and opacity or transparency to the sample point by

45 some transfer function. Second, the mathematical gradient of the sample values at each sample point 20 is estimated by taking the differences between nearby sample points. This gradient is then used in a lighting calculation to determine the brightness of the sample point.

[0067] Figure 2 illustrates the processing of an individual ray. Ray 18 passes through the three dimensional volume data set 10 at some angle, passing near voxels 12. Each sample point is calculated by an interpolation unit 24, and its

50 gradient is calculated by a gradient estimation unit 26. The outputs of these are then applied to a transfer function 28 which assigns color, brightness or intensity, and transparency or opacity to each sample. Finally, the colors, levels of brightness, and transparencies assigned to all of the samples along all of the rays are applied to a compositing unit 30 that mathematically combines their values into pixels depicting the resulting image 32 on image plane 16.

55 **a) Parallel Ray-casting**

[0068] In order to form different points of view of an object, image plane 16 is moved or re-oriented relative to volume data set 10. It is the goal of the field of interactive volume graphics to recalculate images from volume data sets fast

enough from different points of view so that objects appear to move, as if in a motion picture. In addition, the mathematical functions for converting voxels 12 to samples 20 and for accumulating samples into pixels 22 can be modified to provide the appearance of a dynamically changing or transforming three-dimensional object. A typical requirement is to able to recalculate an image thirty or more times per second.

5 [0069] It will be appreciated that the magnitude of recalculation is enormous, even for a small data set of 256^3 voxels. Therefore, most systems implementing the ray-casting technique utilize parallel processing units and cast a multiplicity of rays through the volume at the same time. Figure 3 illustrates some potential ways of implementing parallel ray-casting. In Figure 3A, individual rays 18 are cast through the volume data set 10 independently, stepping through the data set from the front to back accumulating color, intensity, and opacity as they proceed. Parallelism is achieved by assigning separate rays to separate processing units. For example rays labeled "a" through "g" might be assigned to separate processing units for processing in parallel. This is called the "ray-parallel approach."

10 [0070] The problem with the ray-parallel approach is that the same voxel values are needed to process different rays, often at the same time. For example, the voxel labeled 34 in Figure 3A is needed in order to process both ray "c" and ray "d." If two independent processing units proceed at their own pace, then the voxel must be fetched from volume data 15 set 10 at least twice. In general, the value of each individual voxel contributes to several rays that pass near it, so each value of the volume data set needs to be fetched several times by separate processors. Since fetching data from a memory module is a time-consuming operation relative to processing data, this approach is slow and expensive. Moreover, even if voxels were distributed across memory modules, it is likely that several processors would be trying to access the same module at the same time. Thus, memory access is the bottleneck to rendering the volume data set to 20 an image.

25 [0071] In Figure 3B, this problem is partly alleviated by casting several rays in parallel through the volume data set 10, with each ray assigned to a processing unit and with all processing units working in lock step. In this technique, called the "beam parallel" approach, the processing units all together fetch a row or "beam" 36 of voxels at the same time. Each processing unit synthesizes sample points and calculates color, intensity, and transparency from the voxel that it has fetched and from the values fetched by its neighbors to the left and right. Then all processing units step forward to the next beam, each processing corresponding voxels of that beam and sharing values with its neighbors. Then they step forward to the next beam, then the next, etc., until all rays have emerged from the volume data set. This is repeated for other beams, starting on the front face of the volume data set until all of the voxels of have been processed.

30 [0072] In order to avoid memory conflicts, the processing units for separate rays should have independent memory modules, and the volume data set should be distributed across the memory modules. For example, the vertical "slice" 36 of voxels in Figure 3B would be assigned to a separate memory module from the slices on each side. Therefore, its processor could fetch voxels from that memory module while adjacent processing units fetch adjacent voxels from adjacent memory modules concurrently, without memory conflicts. Processing units would, of course, share voxel values with each other in order to synthesize "missed" sample points, calculate normal vectors, and pass rays through the volume at an angle.

35 [0073] This approach works well provided that the rays are roughly parallel to the vertical slices. If, however, the volume is to be viewed from another direction, as illustrated in Figure 3C, then beam parallel processing fails. In this case, a beam of voxels 36 and the vertical slice 38 are parallel to each other. The result is that all of the voxels of the beam are in the same vertical slice, and therefore they are stored in the same memory module. Thus it would not be possible 40 for a multiplicity of parallel processors to access them all at the same time without clashing over access the that memory module.

45 [0074] Some ray-casting system solve this problem by storing three copies of the volume data set at any given time, one for each orientation or major view direction. One copy is partitioned among memory modules in slices front to back, a second copy is partitioned among memory module side to side, and a third is partitioned among memory modules top to bottom. It will be appreciated that this triples the amount of memory needed to store the volume data set, and it also imposes a burden upon applications to keep all three copies consistent with each other.

b) Slice-parallel Ray-casting in Cube-4

50 [0075] Referring now to Figure 4, the problem of requiring three copies of a volume data set is solved in the Cube-4 System by "skewing" the volume data set across memory modules in all three dimensions simultaneously. That is, adjacent voxels are stored in adjacent memory modules, so that no matter which way rays enter the volume data set, adjacent rays pass near adjacent voxels assigned to different memory modules.

55 [0076] Figure 4 illustrates this skewing by showing the detail of a portion of a volume data set near one corner, for a system with four memory modules. Each voxel in the figure is illustrated by a small cube 54, and the patterns on the cubes depict the assignment of voxels to memory modules. As can be seen from the illustration, adjacent voxels on the three visible faces of the volume data set have different shaded patterns and therefore are assigned to different memory modules. It will be appreciated that the same is true for the three faces of the volume data set that are not shown in

Figure 4, that is, the three back faces. In the terminology of the Cube-4 System, this arrangement is called "skewing" and it is the essence of Cube-4 invention.

[0077] The arrangement of voxels among memory modules can be described mathematically. If there are P memory modules and processing units, then a voxel located at position (x, y, z) within the volume data set is assigned to the memory module numbered

$$(x + y + z) \bmod P, \quad (1)$$

where x, y, and z are integers that represent the position of the voxel within the data set in terms of the three dimensions, and where the symbol "mod" is represents the mathematical operation of dividing the quantity on the left by the integer on the right and keeping only the remainder. That is, the memory module number can be obtained via Formula 1 by adding up the three positional coordinates of the voxel, dividing by the number of memory modules P, and taking the remainder, this remainder having values ranging from zero to (P - 1). Although the x, y, and z coordinates of a voxel are typically counted from a designated corner of the data set, it is also possible, without loss of generality, to count from some designated point, this point being referred to as the "origin."

[0078] It will be appreciated from Formula 1 that the voxels of every slice through each of the three dimensions of the volume data set are skewed across memory modules in exactly the same way, but starting with a different memory module. Therefore, if one of the slices were peeled away from any face of the volume data set of Figure 4, it would reveal an identically colored or shaded slice immediately behind it, but with the patterns shifted by one voxel. Moreover, it will be appreciated that voxels of a particular memory module are always bracketed by voxels of the same two other modules, one module on one side and one on the other. Thus, a particular processing unit associated with a memory module has exactly two neighbors.

[0079] This organization has a profound effect on the ability to parallelize the casting of rays. A group of P rays can be cast in any direction through any face of the volume data set, with each ray being assigned to one processing unit, and they can always be processed in parallel without memory conflicts. Each processing unit fetches a voxel near its ray from its own memory module, so that P adjacent voxels are fetch simultaneously and concurrently. Thus, the skewed memory organization enables full parallelism in rendering a volume data set from any view direction.

[0080] Referring now to Figures 5 and 6, the Cube-4 system renders a volume data set a slice at a time, casting all rays through that slice and accumulating the visual characteristics of all sample points within the slice, before proceeding to the next slice. This technique is called "slice parallel" rendering. Figure 5 is a diagrammatic illustration depicting a multiplicity of rays 18 entering the face of a slice 55 of the volume data set. It will be appreciated that in actual practise, there are far more rays than can be illustrated in this simple figure.

[0081] In the slice parallel technique, each ray 18 entering slice 55 is partially rendered. That is, the visual characteristics including color, brightness, and transparency of each ray is assigned so far as possible from the data available in slice 55. Only after all of the rays have accumulated visual characteristics from slice 55 does the Cube-4 system peel away that slice and step forward to the next slice 56. Because of the skewing of voxels across memory, slice 56 is identical to slice 55, but with its memory module assignments shifted by one voxel.

[0082] Figure 6 is an illustration of the actual order of fetching voxels in the Cube-4 system. In this illustration, P is assumed to be four, so there are four memory modules and four processing pipelines. Voxels 54 are fetched from the top row of the slice in groups of P voxels, starting in the upper left corner. Once these P voxels are inserted into the processing pipeline, the next P voxels from the same row are fetched, then the next, etc., until the row is completed. Then the system steps down to the next row, also fetching in groups of P voxels until that row is also completed. This processing is repeated for all rows of slice 55 until that slice is completed. Then processing continues with the next slice 56, also in groups of P voxels starting from its upper left corner.

[0083] It will be appreciated that variations of the Cube-4 algorithm are possible in which processing begins with some other corner or some other designated point, but it always proceeds in groups of P voxels at a time through the beams of a slice and through the slices of a volume data set.

[0084] Figure 7 is an idealized block diagram of the prior art Cube-4 system and illustrates the interconnection of the processing elements and memory modules. In Figure 7, a multiplicity of processing pipelines 40 are each coupled to their own volume memory modules 42. Processing pipelines 40 are coupled to each other via a multiplicity of communication channels 44, each communication channel providing a means for transmitting data in either direction between two processing pipelines. The outputs 46 of the processing elements are coupled to a bus or other mechanism 48 for purpose of conveying pixels of the rendered image to a display surface 50, such as a computer screen. Input voxels are written to volume memory modules 42 via an input bus 52 which is coupled to each module.

[0085] It will be appreciated from Figure 7 that there is no "first" or "master" processing pipeline. All pipeline have equal status and operate in lock step with each other.

[0086] Figure 8 depicts a block diagram of internal elements of Cube-4 processing pipeline, along with a more

detailed view of the communications channels between the pipelines. In the figure are five processing pipelines 40 arranged side-by-side for illustration purpose only. That is, the rightmost processing unit in the figure is connected to the leftmost processing unit so that the entire system form a ring of processing units as in Figure 7. In Figure 8, each memory module 42 is coupled to FIFO storage unit 60 and to trilinear interpolation unit 62 of its own pipeline. Memory module 42 is coupled via communication lines 44 to the trilinear interpolation units 62 of the two neighboring pipelines to the left and to one trilinear interpolation unit 62 of the neighboring pipeline to the right. FIFO storage unit 62 is coupled to one trilinear interpolation unit 62 in its own processing pipeline and via communication lines 44 to one trilinear interpolation unit 52 in the neighboring processing pipeline to the left and to trilinear interpolation units 62 in each of the two neighboring pipelines to the right. By these connections, each processing pipeline can synthesize sample points 20 from the eight surround voxels. FIFO storage units 60 are First-in, First-out storage circuits that provide internal storage necessary to hold copies of voxel values from one beam to the next and one slice to the next.

[0087] Trilinear interpolation unit 62 is coupled both to FIFO storage unit 64 and to shader unit 66. FIFO storage unit 64 is coupled to shader unit 66 of its own pipeline and, via communication lines 44, to shader units 66 of the two neighboring pipelines to the right. Shader unit 66 of a pipeline is also coupled via communication lines 44 to shader units 66 of the nearest neighboring pipelines on either side and also to the shader unit 66 of the second neighboring pipeline to the right.

[0088] The output of shader unit 66 is coupled to compositing unit 68, which is also couple via communication lines 44 to compositing units 68 of the neighboring pipeline to the left and to the three neighboring pipelines to the right.

[0089] Detailed operation of the Cube-4 system, along with the descriptions of the signals passing across communication lines 44 is given in the aforementioned Doctoral Dissertation by Hanspeter Pfister and also in a Master's thesis (i.e., "Diplomarbeit im Fach Informatik") by Urs Kanus and Michael Meissner entitled "Cube-4, a Volume Rendering Architecture for Real-time Visualization of High-resolution Volumetric Datasets," submitted to Eberhard-Karls-Universität Tübingen, in Tübingen, Germany, on September 30, 1996. In general, P voxels are fetched from memory modules 42 and forwarded to a multiplicity of trilinear interpolation units 62 to synthesize samples 20. Since each sample is synthesized from the eight voxels surrounding it, and since these voxels reside in adjacent rows and adjacent slices, it will be appreciated that some voxel values must be "held up" or delayed to wait for others to be fetched. This delay is provided by FIFO storage units 60. Once samples 20 have been synthesized, they are forwarded to shader units 66 where gradients are calculated.

[0090] Each gradient depends upon the values of samples on either side of it and on samples above and below it and on samples in front of and behind it. It will be appreciated that some samples are computed before others, so the earlier sample values must be held up or delayed, just like voxel values, FIFO storage units 64 provide two levels of delay, one level for a single slice and one level for a second slice.

[0091] Finally, after gradients calculated, the color, brightness, and transparency of the samples can be calculated. These visual characteristics are forwarded to compositing units 68 where they are combined with the colors, levels of brightness, and transparencies already accumulated in their respective rays for previously processed voxels and samples. It will be appreciated that a ray may pass through the volume data set at an angle, so that when it emerges from a slice in the vicinity of a voxel, it may enter the next slice in the vicinity of any of nine voxels. These nine voxels are skewed across as many as five memory modules. Therefore, the values of partially accumulated rays must be forwarded to any of five processing pipelines 40 and the compositing units 68, depending upon the view direction, for continued accumulation of color, brightness, and transparency values of additional slices.

[0092] When a ray is finally completed, it is forwarded to the viewing surface via pixel bus 48 for display.

c) Limitations of Memory Access Rates in Cube-4

[0093] Looking in more detail at the assignment of voxels to memory modules, it is possible to see the order in which voxel values are fetched from memory during slice parallel processing. If a volume data set is organized as a cube with N voxels on each edge so that it has a total of N^3 voxels, and if N is evenly divisible by p, then the address of each voxel within its memory module in the Cube-4 system is given by the mathematical formula

$$50 \quad \left[\begin{array}{c} x \\ - \\ p \end{array} \right] + y \frac{N}{p} + z \frac{N^2}{p}, \quad (2)$$

55 where x, y, and z are the integer coordinates of the voxel with respect to a corner or some other origin of the volume data set and where the symbol

$$\begin{bmatrix} x \\ - \\ p \end{bmatrix}$$

5

denotes the result of dividing the integer x by the number P and discarding the remainder.

[0094] Figure 9 illustrates the shading of voxels on the XY face of Figure 4, along with the memory address of each voxel for P equal to four. It will be appreciated from the figure that within any row, groups of P adjacent voxels have the same memory address within their respective memory modules. Moreover, when a processing unit fetches voxels consecutively according to the Cube-4 algorithm, it fetches successive voxels of the same shading from the same row. It can be seen from Figure 9 that these voxels have consecutive memory addresses within their memory modules. More generally, it will be appreciated from Formula 2 that for any slice of the volume data set parallel to the XY face, consecutive voxels have consecutive memory addresses. In theory, it would be possible to use a burst mode DRAM module to fetch these voxels more quickly, provided the view direction was such that rays enter the XY face of the volume.

[0095] However, Figure 10 illustrates the assignment of memory addresses to voxels on the YZ face of the same volume data set with P equals four. On this face, it can be seen from the figure that consecutive voxels having the same shading within the same row differ in their memory addresses by the amount N . Moreover, the last voxel of a row having a given shading and the first voxel of the next row having the same shading differ in their memory addresses by the amount $3xN/4$.

[0096] Therefore, it would not be possible to use burst mode of a DRAM module to fetch consecutive voxels for rays entering the YZ face. A processing unit fetching voxels according to the Cube-4 algorithm would be limited to fetching them in ordinary mode, that is, not in burst mode, at a data rate as slow as if it were fetching memory locations at random.

[0097] It will be appreciated from Formula 2 that in the ZX face, consecutive voxels of any row having the same shading would differ in their memory addresses by N^2 . Therefore, burst mode could not be applied to processing rays entering this face, either.

[0098] More generally, if the volume data set is a rectangular solid with dimensions L , M , and N , where each of L , M , and N is evenly divisible by P , then it has a total of $LxMxN$ voxels. The address of each voxel in the Cube-4 system is given by the formula

$$\begin{bmatrix} x \\ - \\ p \end{bmatrix} + yx \frac{L}{P} + zx \frac{L \times M}{P}, \quad (3)$$

35

[0099] It will be appreciated from Formulas 1 and 3 that consecutive voxels within a row having the same shading on the XY face are stored at consecutive memory addresses, but consecutive voxels within a row having the same shading on the YZ face are stored at addresses differing by L and that consecutive voxels within a row having the same shading on the ZX face are stored at addresses differing by LxM . Thus, burst mode can be used to speed up the fetching of voxels when rays enter the XY face, but it cannot be used when they enter the YZ or ZX faces.

d) Blocking and the Utilization of Burst-mode Dram

45

[0100] Referring now to Figure 11, in order to group voxels in such a way that they can be fetched from consecutive memory addresses, regardless of viewing direction, the subject invention utilizes a technique called blocking. By doing so, it becomes possible to use burst mode to access voxels from DRAM modules for all viewing directions. In this technique, voxels are organized into subcubes or blocks 70, and blocks are skewed across memory modules rather than individual voxels being skewed. The shading of a block in the figure indicates the memory module in which it is stored, with all of the voxels of that block being stored in that same memory module.

[0101] In particular, if each block has B voxels along each of its edges, then the assignment of a voxel with coordinates (x, a, z) is given by the formula.

55

5

$$\left(\left\lfloor \frac{x}{B} \right\rfloor + \left\lfloor \frac{y}{B} \right\rfloor + \left\lfloor \frac{z}{B} \right\rfloor \right) \bmod P, \quad (4)$$

where P is the number of memory modules and processing units and x , y , and z are integer coordinates of the voxel relative to the corner or other origin of the volume data set in each of the three dimensions. That is, the memory module to which voxel (x, y, z) is assigned can be determined by dividing each coordinate by B , throwing away the remainder,

10 taking the sum of these three divisions, then dividing the resulting sum by P and taking the remainder. This is the same formula described by Lichtermann in the aforementioned description of the DIV²A system.

[0102] Blocks are numbered within the volume data set in the subject invention in the same way as voxels are numbered in de Cube-4 system, that is by counting blocks in each of the three dimensions from the corner or other origin. It will be appreciated from Formula 4 that a voxel at position (x, y, z) is stored in a block with block coordinates (B_x, B_y, B_z) given by the formulas.

20

$$B_x = \left\lfloor \frac{x}{B} \right\rfloor, \quad B_y = \left\lfloor \frac{y}{B} \right\rfloor, \quad B_z = \left\lfloor \frac{z}{B} \right\rfloor \quad (5)$$

[0103] If the volume data set represents a cube with N^3 voxels, and if $P \times B$ evenly divides N , the number of voxels on 25 a side of the cubic data set, then the starting address of the block with coordinates (B_x, B_y, B_z) within its memory module is given by the formula

30

$$\frac{B_x \times B^3 + B_y \times N \times B^2 + B_z \times N^2 \times B}{P} \quad (6)$$

35 [0104] Within each block, the voxels are stored at consecutive memory addresses. It will be appreciated that many possible arrangements of voxels within a block are possible. In one embodiment, voxel memory addresses relative the beginning of the block are given by the formula

40

$$x \bmod B + B * (y \bmod B) + B^2 * (z \bmod B). \quad (7)$$

45

[0105] That is, the position of voxel (x, y, z) within its block can be found by taking the remainders from x , y , and z after dividing by B , then adding the remainder from x to B times the remainder from y and then adding that to B^2 times the remainder from z . It will be appreciated that Formula 7 describes consecutive locations with a range of B^3 memory addresses, where B^3 is the number of voxels in a block.

[0106] Referring now to Figure 12, a diagrammatic illustration is of the pipeline processor 40 of one embodiment of the subject invention along with its associated memory module 42. Like Cube-4, the subject invention comprises a multiplicity of processing pipelines 40 and memory modules 42 connected in a ring, as illustrated in Figure 7. Referring again to Figure 12, memory module 42 is coupled to two block buffers 72, each of which has capacity to store B^3 voxels, where B is the number of voxels on the edge of a block. Each block buffer 72 is coupled both to interpolation unit 82 and to two tri-state interfaces 74. Each tri-state interface 74 is coupled to a voxel communication line 76, one being coupled to the nearest neighboring pipeline 40 in the clockwise direction around the ring and the other being coupled to the nearest neighboring pipeline in the counter-clockwise direction around the ring.

[0107] It will be appreciated that in electronic design, a tristate interface is one which serves as either an input or output interface. In particular, in a semiconductor implementation of the present embodiment, the pins connecting tri-state interface 74 to communication line 76 are both input and output pins. Therefore, each voxel communication line 76 can

carry data in either direction, so that processing pipeline 40 can either receive data from or transmit data to either of its nearest neighbors. In this embodiment, both voxel communication lines 76 carry data in the same direction around the ring, that is, either both are configured for clockwise signaling or both are configured for counterclockwise signaling at any given instant.

- 5 [0108] Tri-state interface 74 are also coupled to beam FIFO storage unit 78, to slice FIFO storage unit 80 and to optional delay unit 73. Beam FIFO storage unit 78, slice FIFO storage unit 80, and optional delay unit 73 are all coupled to interpolation unit 82. For rendering a volume data set L voxels wide, M voxels high, and N voxels deep, beam FIFO storage unit 78 is configured to hold $L \div (B \times P)$ elements, where each element is an array of $(B + 1)^2$ voxels. Likewise, slice FIFO storage unit 80 is configured to hold $(L \times M) \div (B^2 \times P)$ elements, where each element is an array of an array of 10 $Bx(B + 1)$ samples. Optional delay unit 73 is configured to hold B^2 voxels and to delay them either zero fundamental cycles or B^3 fundamental cycles, depending upon whether the pipeline is at the left end of its partial beam or not. As will be shown below, beam and slice FIFO storage units 78 and 80 hold voxels forwarded from immediately above and in front of the block being processed, respectively. Optional delay unit 73 holds voxels forwarded from the pipeline immediately to the left.
- 15 [0109] Interpolation unit 82 calculates the values of sample points based on the immediately surrounding voxels. In general, to calculate B^3 sample points, $(B + 1)^3$ voxel values are needed. These are obtained from the B^3 voxels read from voxel memory 42 into block buffer 72, plus an array of B^2 voxels from optional delay unit 73, an array of $(B + 1)^2$ voxels from beam FIFO storage unit 78, and an array of $Bx(B + 1)$ voxels from slice FIFO storage unit 80.
- 20 [0110] Interpolation unit 82 is coupled to gradient estimation and shading unit 92 and to tri-state interfaces 84. Tri-state interfaces 84 are coupled to sample communication lines 86, which are in turn coupled to the nearest neighbor pipelines in the clockwise and counterclockwise directions, respectively. Like voxel communication lines 76, sample communication lines 86 are bidirectional and may carry sample data in either direction around the ring. Tri-state interfaces 84 are also coupled to beam FIFO storage unit 88, slice FIFO storage unit 90, and optional delay unit 83. Optional delay unit 83 and beam and slice FIFO storage units 88 and 90 are all coupled to gradient estimation and shading unit 92.
- 25 [0111] For rendering a volume data set L voxels wide, M voxels high, and N voxels deep, beam FIFO storage unit 88 is configured to hold $L \div (B \times P)$ elements, where each element is an array of $2x(B + 2)^2$ samples. Likewise, slice FIFO storage unit 80 is configured to hold $(L \times M) \div (B^2 \times P)$ elements, where each element is an array of $2xBx(B + 2)$ samples. Optional delay unit 83 is configured to hold B^2 sample values for a delay of either zero fundamental cycles or B^3 fundamental cycles, depending upon whether the pipeline is at the left end of its partial beam or not. As will be shown below, beam and slice FIFO storage units 88 and 90 hold samples forwarded from immediately above and in front of the block being processed, respectively. Optional delay unit 83 holds samples forwarded from the pipeline immediately to the left.
- 30 [0112] Gradient estimation and shading unit 92 is coupled directly to compositing unit 102. Compositing unit 102 is coupled to tri-state interfaces 94, which in turn are coupled to composition element communication lines 96. As with voxel communication lines 76 and sample communication lines 86, composition element communication lines 96 are bi-directional communication lines to the nearest neighboring pipeline in each of the clockwise and counterclockwise direction around the ring. Tri-state interfaces 94 are also beam FIFO storage unit 98, slice FIFO storage unit 100, and optional delay unit 93. Beam FIFO storage unit 98, slice FIFO storage unit 100, and optional delay unit 93 are all coupled to compositing unit 102. Finally, compositing unit 102 is coupled to pixel output bus 48, which is in turn coupled to a viewing surface such as a computer screen.
- 35 [0113] For rendering a volume data set L voxels wide, M voxels high, and N voxels deep, beam FIFO storage unit 98 is configured to hold $L \div (B \times P)$ elements, where each element is an array of $(B + 1)^2$ pixel values of partially accumulated rays, that is, visual characteristics containing color, opacity, and depth information. Likewise, slice FIFO storage unit 100 is configured to hold $(L \times M) \div (B^2 \times P)$ elements, where each element is an array of $Bx(B + 1)$ pixel values of 40 partially accumulated rays. Optional delay unit 93 is configured to hold B^2 pixel values of partially accumulated rays with a delay of either zero fundamental cycles or B^2 fundamental cycles. As will be shown below, beam and slice FIFO storage units 98 and 100 hold pixel values of partially accumulated rays forwarded from immediately above and in front of the block being processed, respectively. Optional delay unit 93 holds pixel values of partially accumulated rays from the pipeline immediately to the left.
- 45 [0114] In other words, in the present embodiment of the subject invention, a processing unit 40 comprises four major functional stages connected together in pipeline fashion, namely, a block buffering stages, an interpolation stage, a gradient estimation and shading stage, and a compositing stage. Each stage is separated from the next by a pair of bi-directional communication lines to the neighboring pipelines and by beam and slice FIFO storage units capable of holding values forwarded from the previous beam and the previous slice.

55 e) Method of Operation

- [0115] The method of operation of this embodiment of the subject invention will now be described. Referring to Figure

13, the view direction is determined. A single ray 110 is cast perpendicularly from the view surface through the volume data set, that is from image 10, so that it strikes the center of face 12 that is nearest and most nearly perpendicular to the view surface. In general, the ray will strike the view surface at some angle 116 less than 45 degrees from the normal vector of face 112, that is, from a line perpendicular to face 112. It will be appreciated that if the angle 116 is greater than 45 degrees, a different face of the volume data set would be nearer and more nearly perpendicular to the ray 110. If the angle 116 to normal vector line 114 is exactly 45 degrees, then either of two view surfaces can be chosen arbitrarily. Moreover, if ray 110 strikes a corner of the volume data set, then angle 116 will be 45 degrees from each of three normal vectors will be 45 degrees, and any of the three faces can be chosen arbitrarily.

5 [0116] Having selected a face 112, ray 110 is projected onto the face, making a "shadow" 118 of the ray. In general, 10 this shadow will land in one of the four quadrants of the face. The quadrant 120 containing shadow 118 will be the selected quadrant. If shadow 118 lands on a line between two quadrants, then either quadrant can be selected. If shadow 118 is a point exactly in the center of face 112, then ray 110 is perpendicular to the face and any quadrant may be selected.

15 [0117] Having selected a quadrant 120 of a face 112, the volume data set may now be rotated in three dimensions so that face 112 is at the "front" and quadrant 120 is in the upper left corner. It will be appreciated that in "rotating" the volume data set, no data has to be moved. Instead, an appropriate transformation matrix can be applied to voxel and block coordinates to translate these coordinates into coordinate system in which the corner of the selected quadrant is the origin and is in the upper left corner of the front face. The theory of transformation matrices is explained in graphics textbooks, including the aforementioned reference by J. Foley, et al.

20 [0118] In the following discussion, coordinates relative to the volume data set itself are denoted as x, y, and z, while coordinates relative to the selected quadrant 120 are denoted u, v, and w. These are called "rendering coordinates." The terms "left," "right," "above," "below," "back," and "front" are defined in terms in rendering coordinates as follows:

25	"Left"	"in the direction of decreasing values of u."
----	--------	---

30	"Right"	"in the direction of increasing values of u."
35	"Above" and "top"	"in the direction of decreasing values of v."
40	"Below" and "bottom"	"in the direction of increasing values of v."
45	"Front"	"in the direction of decreasing values of w."
50	"Back"	"in the direction of increasing values of w."

55 [0119] Moreover, the front, left, top corner of the volume data set in rendering coordinates is designated as the "origin," that is, the voxel with $(u, v, w) = (0, 0, 0)$. In rendering coordinates, rays always pass through a volume from front to back and in a downward and rightward direction, unless they happen to be perpendicular to the face.

[0120] It will be appreciated from the definition of block skewing in Formula 4 that the association of the terms "left,"

"right," "above," "below," "front," and "back" with particular neighboring pipelines depends upon the viewing direction. For one viewing direction, the pipelines in front, to the left, and above may all be the pipeline in the counterclockwise direction in Figure 7, while in other viewing directions, some or all of them may be the pipeline in the clockwise direction in Figure 7.

5 [0121] In the current embodiment of the subject invention, the processing order is exactly that of Figure 6 except that it references blocks, not individual voxels. That is, processing begins at the origin in rendering coordinates and proceeds from left to the right in groups of P blocks across each beam of blocks, then beam by beam down the slice blocks, and then slice by slice from the front of the volume to the back. As the precessing pipelines step across the volume in groups of P blocks, rays will always exit blocks in the direction still to be processed, that is, either to the back, the bottom, or the right. In all cases, already processed data will come from above, to the front, and to the left of a block currently being processed.

10 [0122] Referring again to Figure 12, the fundamental processing cycle in the current embodiment of the subject invention is the cycle time of reading one voxel from DRAM memory in burst mode. These fundamental cycles are group into block cycles of B^3 fundamental cycles each. At the beginning of a block cycle, B^3 voxels are fetched from consecutive 15 addresses of memory module 42, starting at the beginning of the block under consideration and continuing for B^3 fundamental cycles. Voxels are fetched into one of the two block buffers 72. During the next block cycle, those B^3 voxels will be processed while a new block of B^3 voxels is fetched into the other block buffer 72. Then during the following block cycle, the roles of the two buffers are reversed again, in an application of the familiar technique of "double buffering."

15 [0123] Referring now to Figure 14, in both Cube-4 and the subject invention, the spacing of rays 18 is determined not by pixels 22 on image plane 16, but by base pixels 130 on the base plane 132. Figure 14 depicts a two-dimensional illustration of a three-dimensional volume data set and image plane as in Figure 1. The "base plane" of Figure 14 is a mathematical plane parallel the selected face 112 of the volume data set and passing through the origin $(u, v, w) = (0, 0, 0)$ in rendering coordinates. "Base pixels" 130 of base plane 132 are coterminous with voxels on face 112, and they extend in all directions with the same spacing as voxels. Rays 18 are cast in a direction perpendicular to image plane 16 but passing through the exact centers of base pixels 130 in base plane 132. The resulting image is then rendered into the base plane, not the image plane. It will be appreciated that, in general, rays passing through base pixels will not line up exactly with pixels 22 of image plane 16. Therefore, a postprocessing step is required to "warp" the base plane image into a final image.

20 [0124] It will also be appreciated that for rays 18 that are parallel to each other, sample points are offset in space from their neighboring voxels by the same amount whenever they lie in the same plane parallel to the base plane. This simplifies the Cube-4 algorithm considerably. In particular, it means that adjacent sample points are surrounded by adjacent groups of eight neighboring voxels, with four of those eight being shared between the two sample points.

25 [0125] The flow of data among pipelines during the operation of the current embodiment will now be described. First, an array of B^3 sample values along rays 18 is calculated from the B^3 voxels of a block plus other voxels forwarded from neighboring pipelines. Since samples are interpolated from their nearest voxels, it will be appreciated that it takes an array of $(B + 1)^3$ voxels to generate B^3 samples. Second, an array of B^3 gradients is calculated and pixels values representing colors, brightness or shading levels, and transparency levels are assigned. Since it requires the values of samples on all sides of a given sample in order to estimate its gradient, a total of $(B + 2)^3$ samples is needed to generate B^3 gradients and pixel values. Finally, the B^3 pixel values are composited with previously accumulated pixel values to form partially rendered rays. This also requires an array of $(B + 1)^3$ pixel values to accumulate the visual characteristics of the rays passing through a block.

30 [0126] Figure 15 illustrates three views of a three-dimensional array of $(B + 1)^3$ voxels needed by a pipeline to calculate a block of B^3 sample points. Figure 15A represents a cross-section of the array for values of $w > 0$, that is, all voxels except the front face of the array. Figure 15B represents the right face of the array. Figure 15C depicts a perspective view of the three dimensional array from a view below, in front of, and to the right of the array.

35 [0127] The voxels of the cubic array in Figure 15 come from four sources. A block of B^3 voxels 140 is fetched from volume memory into block buffer 72. An array of B^2 voxels 142 is forwarded from the pipeline on the left of the current block via communication lines 76 and optional delay unit 73. An array of $(B + 1)^2$ voxels 144 is taken from the output side of beam FIFO storage unit 78, and an array of $Bx(B + 1)$ voxels 146 is taken from the output side of slice FIFO storage unit 80. It will be appreciated that the total of these four groups of voxels is $(B + 1)^3$. The array 150 of B^3 samples, represented in the figure by crosses, is calculated by trilinear interpolation or some other mathematical function. It will be appreciated that, in general, the array 150 of B^3 samples calculated by this process is offset to the left, front, and above the array 140 of B^3 voxels originally fetched from volume memory 42 via block buffer 72. The amount of the offset is always less than the spacing between voxels, but it may be zero in the case of view directions that are perpendicular to one of the axes of the rendering coordinates.

40 [0128] Since P processing pipelines are operating in parallel, voxel array 142 will, in general, be the right face of the block of B^3 voxels currently being processed immediately to the left. Therefore, as voxels of a block are being fetched into a block buffer 72, its rightmost B^2 voxels must immediately be forwarded to the processing element on the right and

inserted into optional delay unit 73, then forwarded to interpolation unit 82. This forwarding must be completed before the voxels are needed for calculating the sample points 150 at the left most edge of block 140. The exception is when block 140 is the leftmost block of a partial beam. In this case, the block to the left was read during the preceding block cycle, so the array of voxels 142 needs to be delayed by one block cycle, that is, by B^3 fundamental cycles. This delay is represented by optional delay unit 73 in Figure 12. In the case that a pipeline is at the left end of its partial beam, the delay value is set to B^3 cycles, but otherwise it is set to zero, meaning no delay at all.

[0129] In addition to forwarding the rightmost face of block 140 for immediate use by the processing pipeline to the right, it is also necessary to prepare arrays for the processing pipelines below and behind block 140. Referring now to Figure 16, array 242 mimics array 142. This will be needed during the processing of the next beam, that is, $L-(B \times P)$ block cycles later. This array is formed from the bottom face of block 140, the bottom row of array 142, and the bottom row of array 146. It is forwarded to the processing pipeline of the block below for storage in its beam FIFO storage unit 78. Likewise, an array 246 mimicking array 146 must be prepared for the processing pipeline of the block behind. This is formed from the back face of block 140 and the back vertical row of array 142. It is forwarded to the slice FIFO storage unit 80 of the processing pipeline of the block behind, to be ready for use one slice later, that is $L \times M-(B^2 \times P)$ block cycles later.

[0130] The calculation of gradients and the compositing of rays follows roughly the same pattern. Interpolation unit 82 produces an array of B^3 samples. In general, as illustrated in Figure 15, these are offset slightly above, to the left of, and in front of the B^3 voxels of block 140. In order to calculate B^3 gradients from these samples, a cubic array of $(B + 2)^3$ samples is required. This is because each gradient is calculated by taking the central differences or some other mathematical function of the adjacent samples in each of the three dimensions.

[0131] Referring now to Figure 17, gradients 158 can be calculated at sample points that are not at the boundary of the $(B + 2)^3$ group of samples. The voxels of the original block 140 are illustrated by dots. Samples calculated by interpolation unit 82 are illustrated by crosses. The original group calculated as part of processing block 140 is the B^3 array 150 of samples. In addition, an array 152 of $2 \times B^2$ samples is needed from the block immediately to the left, an array 154 of $2 \times (B + 2)^2$ samples is needed from the processing of the block above, and an array 156 of $2 \times B \times (B + 2)$ samples is needed from the block immediately in front. As with voxel array 142, array 152 is being calculated by the processing pipeline immediately to the left during the same pipeline cycle, unless block 140 is at the left of a partial beam. Therefore, sample array 152 is forwarded to interpolation unit 82 optional delay unit 83. The delay value is set to zero except when the pipeline is at the left end of its partial beam, in which case the delay value is set to B^3 fundamental cycles, that is, one block cycle. Arrays 154 and 156, by contrast, are obtained from the beam and slice FIFO storage units 88 and 90, respectively.

[0132] Likewise, following the calculation of samples, the interpolation unit 82 must therefore forward arrays mimicking arrays 154 and 156 to the beam and slice FIFO storage units 88 and 90, respectively, for processing in the next beam and slice.

[0133] It will be appreciated that the processing pipeline that originally fetched block 140 of voxels calculates gradients on samples that are offset to the left, above, and in front by more than the spacing of one voxel. That is, that it calculates gradients for some of the samples synthesized on earlier block cycles. The reason for this is that no processing unit can calculate a gradient that is dependent upon voxels and samples that are synthesized later. In particular, the processing unit at the right end of a partial beam cannot calculate gradients for the B^2 samples nearest the right face of its block. Moreover, no processing pipeline can calculate gradients for the samples on the bottom face of its block. Therefore, these have to be calculated later, but to compensate, the processing pipeline must calculate previously uncalculated gradients for samples above, to the left, and in front of its block.

[0134] Following the calculation of gradients, pixel values representing the color, brightness, and transparency or opacity of a sample can be assigned. These are then passed by gradient estimate and shader unit 92 to the final set of stages for compositing. These follow the pattern of Figure 15, but offset in sample space to the positions of the calculated gradients in Figure 17. B^3 pixel values are forwarded directly from gradient estimation and shading unit 92 to compositing unit 102. An array of pixel values of partially accumulated rays mimicking voxel array 142 is forwarded to the processing pipeline to the right, where it is inserted into optional delay unit 93. Likewise, array of compositing elements mimicking voxel arrays 144 and 146 are forwarded to the beam and slice FIFO storage units 98 and 100, respectively, of the neighboring pipelines to the rear and below. As before, these arrays are formed from the bottom and rear slices of the B^3 compositing elements calculated from the gradients in the same block cycle, plus the bottom rows of voxels obtained from the left neighbor and from the slice FIFO storage unit 100.

[0135] In this way, all of the voxels can be processed in the groups of P blocks at a time, stepping right across a beam of blocks, then stepping beams down a slice of blocks, and stepping slices through the volume data set.

[0136] It will be observed that in some embodiments, the processing of each block within a pipeline is carried out in serial fashion in B^3 fundamental cycles. When accessing volume memory at burst mode rate of 125 megahertz or 133.3 megahertz, the length of a fundamental cycle is 8 nanoseconds or 7.5 nanoseconds respectively. This is very demanding upon the designer of the circuitry that calculates samples, estimates gradients, and composites pixel values in par-

tially accumulated rays. Therefore, in preferred embodiments, processing with a pipeline is carried out in parallel by a multiplicity of processing units operating at a slower rate. For example, if B equals 8, then processing can be carried out by a single set of processing stages operating with 7.5 nanosecond to 8 nanosecond cycle times, or by two sets of processing stages operating at 15 to 16 nanosecond cycle times, or by four sets of processing stages operating at 30 to 32 nanosecond cycle times, or by eight sets of processing stages operating at 60 to 64 nanosecond processing times, or by even more stages. It will be appreciated that the selection of the number of processing stages is an implementation choice left to a practitioner skilled in the art of electronic circuit design.

5 [0137] It will be appreciated from the foregoing description that the introduction of blocking and the associated changes to the architecture and processing order of voxels have the effect of making it possible for a system based on
10 Cube-4 to utilize burst mode for accessing DRAM. That is, it becomes possible in the subject invention to implement the large amounts of memory required to hold volume data sets utilizing inexpensive, readily available DRAM devices. This leads to substantial savings in cost, size, and power over previously implemented real-time volume rendering systems, and it makes it possible to implement practical and affordable real-time volume rendering systems for personal and desktop computing environments.
15 [0138] It will also be appreciated that as semiconductor technology advances and as it become possible to combine processing logic and DRAM on the same device or chip, the same architectural changes needed to enable burst mode DRAM access will be needed to enable direct, on-chip access by a processing pipeline to volume memory in an efficient and effective manner.

20 f) Communication between processing pipelines

[0139] From Figure 12, and the discussion above, it can be seen that three kinds of data must be passed from one pipeline to its neighbor in the current embodiment. These are voxels, samples, and partially accumulated pixel values. These are transmitted between pipelines via communication lines 76, 86, and 96, respectively. Moreover, for each kind
25 of data there are two kinds of FIFO storage units, namely, beam FIFO storage units 78, 88, and 98 for voxels, samples, and pixels, respectively, and slice FIFO storage units 80, 90, and 100, respectively.
[0140] During every block cycle, B^3 voxels are fetched from volume memory 42. At the same time, B^2 voxels of array 142, $(B + 1)^2$ voxels of array 144, and $Bx(B + 1)$ voxels of array 146 must be transmitted between two pipelines. That is,

$$30 \quad B^2 + (B + 1)^2 + Bx(B + 1) = 3B^2 + 3B + 1 \\ = (B + 1)^3 - B^3 \quad (5)$$

35 voxels must be transmitted.

[0141] Similarly, during every block cycle, B^3 samples are obtained from the sample calculation stage of the pipeline, but B^2 samples of array 152, $2x(B + 2)^2$ of array 154, and $2xBx(B + 2)$ samples of array 156 are transmitted between pipelines. This is, a total of

$$40 \quad 2xB^2 + 2x(B + 2)^2 + 2xBx(B + 2) = 6B^2 + 12B + 8 \\ = (b + s)^3 - B^3 \quad (6)$$

45 samples must be transmitted.

[0142] Finally, during every block cycle, B^3 pixel values are calculated within the pipeline, but B^2 pixels values representing partially accumulated rays are needed from the left, $(B + 1)^2$ pixels values representing partially accumulated rays are needed from above, and $Bx(B + 1)$ pixel values are needed representing partially accumulated rays from the front. Thus, a total of

$$50 \quad B^2 + (B + 1)^2 + Bx(B + 1) = 3B^2 + 3B + 1 \\ = (B + 1)^3 - B^3 \quad (7)$$

55 pixels values must be transmitted between pipelines. It will be seen from Equations 5, 6, and 7 that the total number of items of data that must be transmitted between pipelines for each B^3 block of voxels is approximately proportional to B^2 .

[0143] The following table displays values for Equations 5, 6, and 7 for block sizes ranging from 2 to 16.

<u>B</u>	<u>B^3</u>	<u>$(B + 1)^3 - B^3$</u>	<u>$(B + 2)^3 - B^3$</u>
2	8	19	56
4	64	61	152
8	512	217	488
16	4096	817	1736

15

[0144] As can be seen from the table, as B grows, B^3 grows rapidly but Equations 5, 6, and 7 grow much more slowly. This is because for each cubic array of data fetched from memory or a previous stage, only a quadric number of voxels need to be transmitted between pipelines. For example, if a block has two voxels on each edge, then 2^3 or eight voxels must be fetched during a block cycle, but 19 voxels, 56 samples, and 19 pixels must be transmitted to the neighboring pipelines during the same block cycle. This is almost twelve times as much information communicated as fetched.

[0145] On the other hand, for B = 8, for each 512 voxels fetched during a block cycle, 217 voxels, 488 samples, and 217 pixels must be transmitted to neighboring pipelines. The ratio of communicated values to fetched values in this case is about 1.8.

[0146] Therefore, a side effect of blocking is the reduction in the amount of information that must be transmitted between pipelines. This has the added benefit to the design of a semiconductor implementation of a processing pipeline because a reduction in the amount of communication is a reduction in the number of pins.

[0147] It is also possible to reduce the bandwidths of pins rather than their number.

[0148] From Figure 8, it will be appreciated that the prior art Cube-4 system requires communication lines between adjacent processing pipelines, pipelines that are a distance of two from each other, and in the case of the compositing unit 68, pipelines that are a distance of three from each other. The total number of pins required is slightly under two hundred for eight-bit voxels and nearly four hundred for sixteen-bit voxels. By contrast, in the current embodiment with B = 8, and assuming 16-bit voxels and samples and 48-bit pixels, the total number of pins required is 192, that is, 96 pins leading to the pipeline on each side.

[0149] It will be appreciated from the method of block skewing in the subject invention that once rendering coordinates have been chosen, a ray may pass through a particular block will exit that block and may enter any one of seven other blocks, namely the three adjacent blocks to the right, below, or behind the given block, the three blocks with edges adjacent to the right bottom, right back, and bottom back edges of the given block, or the block with a top, left, front corner adjacent to the bottom, right, rear corner of the given block. These seven blocks are processed by at least three different processing modules, but possibly as many as five.

[0150] Whereas in the Cube-4 system, each processing pipeline requires direct connections to all five, in the subject invention this is not necessary. Instead, all communication necessary to forward voxel and sample values and partially accumulated rays is accomplished by the forwarding of arrays of these values to nearest neighbors. Values needed by more distant pipelines, that is by pipelines that are not nearest neighbors, are forwarded in several steps, but will always arrive that the destination pipeline in time.

g) Sectioning

[0151] In the subject invention, each processing pipeline requires internal storage or memory to hold data values transmitted from one pipeline to the next. These data values are retained this memory until needed for processing a subsequent beam or subsequent slice of blocks. In the current embodiment, this internal memory takes the form of beam FIFO storage units 78, 88, and 98 and slice FIFO storage units 80, 90, and 100. Each FIFO storage unit is an internal memory unit that implements the well-known technique of First-In, First-Out memory management. That is, new data is always written to unused memory locations. When previously stored data is read, its memory locations become unused again and are available for new data. The control of a FIFO storage unit guarantees that data items can only be read in the order that they are written.

[0152] In the current embodiment, each beam FIFO storage unit requires a capacity to store data for L/(B x P) blocks to be processed later, where L is the width of the volume data set being rendered, B is the number of voxels on the edge

of a block, and P is the number of processing pipelines in the volume rendering system. The amount of data stored per block in FIFO storage units 78 and 98 are arrays of size $(B + 1)^2$ data elements, the amount of data stored per block in FIFO storage unit 88 is an array of size $2 \times (B + 2)^2$ data elements. Thus the total amount of internal storage for beam FIFO storage units is approximately $L \times B \times P$.

- 5 [0153] In the case of slice FIFO storage units 80, 90, and 100, each requires a capacity for $L \times M \times (B^2 \times P)$ blocks to be processed later, where M is the height of the volume data set. The amount of data stored per block in FIFO storage units 80 and 100 arrays of size $B \times (B + 1)$ data elements, the amount of data stored per block in FIFO storage unit 88 is an array of size $2 \times B \times (B + 2)$ data elements. Thus the total amount of internal storage for beam FIFO storage units is approximately $L \times M \times P$.
- 10 [0154] In other words, the storage capacities of beam FIFO storage units must be proportional to the width of the volume data set, while the capacities of slice FIFO storage units must be proportional to the area of the face of the volume data. In the case of cubic volume data sets, that is volume data sets with the same number of elements on each edge, these storage requirements are independent of view direction. But in the cases of general volume data sets shaped like rectangular solids of arbitrary proportions, the storage requirements vary with view direction.
- 15 [0155] It will be appreciated that in the prior art Cube-4 system and in other embodiments of the subject invention, the internal storage requirements are approximately the same. That is, beam storage is proportional to the width of the volume data set, and slice storage is proportional to the area of the face of the volume data set. It will also be appreciated that the storage requirements for slice FIFO units dominates all others.
- 20 [0156] Moreover, the total amount of storage can be very large. In the current embodiment, with voxels and samples needing 16 bits of storage and partially accumulated pixels need 32 bits of storage, more than 1.5 megabits of internal storage are required per pipeline to render a 256^3 data set with four processing pipelines. In current semiconductor technology suitable for implementing processing elements, this would result in a very large and therefore very expensive chip.
- 25 [0157] To alleviate this problem, the subject invention renders a volume data set in pieces or "sections" in such a way as to reduce the area of the face of each section to a more manageable value. This reduces the amount of internal storage needed because each section is rendered as a separate volume data set. Figure 18 depicts three example volume data sets in rendering coordinates, each to be rendered from view direction 110. Figure 18A is partitioned into equal horizontal sections 200. Each section is a "slab" of data from front to back in the rendering coordinates. The amount of internal storage required for slice FIFO units is proportional to the area of each section face 202.
- 30 [0158] In Figure 18B, the volume data set 10 is partitioned in both the u and v dimensions into rectangular sections that extend from front to back. Again, the amount of slice FIFO storage required for rendering each section is proportional to the area of the face. In Figure 18C, the volume data set is a long, thin rectangular solid to be rendered from its long side. In the case, the data set is partitioned into sections with square faces side-by-side. In all three cases, the amount of internal storage required for rendering is reduced to an amount proportional to the areas of the faces 202 of the sections.
- 35 [0159] If the view direction 110 is exactly perpendicular to each section face 202, then the images resulting from rendering each section can be placed side-by-side or one above the other to produce a correctly rendered image of the entire volume data set. That is, in the case a perpendicular view direction, rendering in sections is exactly equivalent to rendering the data set as a whole.
- 40 [0160] However, if the view direction is not perpendicular to each face, then some rays will pass through the boundaries 204 between sections. In this case, partially composited or accumulated rays that exit the side or bottom of a section and into an adjacent section must be stored temporarily until the rendering of that section is complete. They must then be used to continue the accumulation of rays in the adjacent sections.
- 45 [0161] In the subject invention, partially accumulated rays are stored outside of rendering pipelines in external storage modules. Referring now to Figure 19, a modification of the lower portion of Figure 12 is shown. In this modification, compositing unit 102 is coupled to external storage module 104 via tri-state unit 106. Tri-state units 94 connecting pixel communication lines 96 are also coupled to tri-state interface 106. External storage module 104 is then coupled to rendering output 48.
- 50 [0162] During rendering operation, when compositing unit 102 sends a partially accumulated pixel value to its neighboring pipeline below or to the right, if that pixel value represents a ray leaving right or bottom side of the section currently being rendered and if that ray will enter the left or top side of an adjacent section, then the neighboring pipeline writes the pixel value to external storage module 104 via tri-state interface 106. Then, when rendering the adjacent section, pixel values of partially accumulated rays that enter the section from the left or above must be read from the external storage unit 104 instead of from optional delay 93 or beam FIFO storage 98. Pixel values representing rays that leave the entire volume data set are also written to external storage module 104, then forwarded to pixel output lines 48.
- 55 [0163] In the subject invention, external storage is implemented utilizing burst mode DRAM chips. The amount of storage required is proportional to the areas of the bottom and right faces forming the boundaries of sections. For example, if a cubic data set with N voxels on each edge is divided into horizontal sections as in Figure 18A, then each boundary

204 will have at most N^2 pixels. The actual number of pixels needing to be stored between sections depends upon the viewing angle. Perpendicular viewing directions require no pixels to be stored externally, which viewing angles at 45 degrees to the boundary of a section require that N^2 pixels be stored.

5 [0164] It will be appreciated that to correctly accumulate rays at the boundaries of sections 204, sample points must be calculated between the right or bottom plane of voxels of one section and the top or left plane of the adjacent section. The calculation of these samples requires voxel values from both sections. Similarly, gradients must be estimated for sample valued both between sections and for the sample values within each section nearest the boundary 204. This estimation requires sample values from both sections.

10 [0165] In some embodiments of the subject invention, these voxel and sample values required for the calculations near the section boundaries are stored in external storage 104 along with pixel values of partially accumulated rays. In preferred embodiments, however, these voxel and sample values are obtained by re-reading and re-rendering the voxels near the boundary of the section. This has the effect of refilling the beam FIFO storage units with the appropriate intermediate values needed to calculate the samples and gradients near a section boundary 204.

15 [0166] It will be appreciated that the technique of sectioning can be utilized for rendering volume data sets that are too large to fit into volume memory all at once. A large data set would be subdivided into sections, each of which is small enough to fit into volume memory and each of which has a face that is small enough to render within the limits of internal storage. Then each section is rendered separately, and its rendered image is accumulated in external storage modules 104. After a section is rendered, it is replaced in volume memory with an adjacent section which is then rendered separately. Pixel values of partially accumulated rays are passed from one section to the next via the same mechanism as described above, including the re-reading and re-processing of voxels near the boundary. This process is repeated until the entire volume data set is rendered. It will be appreciated that in cases of very large data sets, it is impractical to achieve the full real-time rendering rate of 30 frames per second.

b) Mini-blocking

25 [0167] In the subject invention as described thus far, voxels are read from voxel memory a block at a time. That is, each read operation fetches B^3 voxels in one block cycle. This is more than needed to re-process the voxels near the boundary of a section. For example, if B equals 8, then 512 voxels are read in a block cycle. However, only two or three planes of voxels are needed along the boundary for reprocessing, that is, only 128 or 192 voxels. The requirement to read 512 voxels at a time in order to process only 128 or 192 voxels is a waste of time.

30 [0168] To reduce this waste, the subject invention utilizes a technique called "mini-blocking." In this technique, voxels are stored in with a block in smaller blocks of "mini-blocks" of size $2 \times 2 \times 2$ voxels each, that is a total of eight voxels in a mini-block. The entire block is then a cubic array of mini-blocks. Each mini-block is stored at consecutive locations in volume memory, so that it can be fetched in burst mode. The mini-block size is chosen to be at least as large as the minimum burst size of some burst mode DRAM products. Therefore, the mini-blocks of a block can be read from volume memory in any order, and a subset of a block can be read whenever appropriate.

35 [0169] This technique is utilized when re-reading the voxels near the boundary of a section. Only as many mini-blocks as containing the voxels needed for re-processing are fetched.

40 [0170] It will also be appreciated that the sizes of block buffers can be reduced slightly by fetching mini-blocks in the order that voxels are processed within a block. That is, the rendering of a block begins at one side as soon as those voxels have been fetched. So long as fetching proceeds as fast as the processing of individual voxels, the two can proceed in lock step. Then, new voxels can be fetched into buffer areas for which processing is completed. This reduces the total amount of block buffer storage 72 needed for rendering.

45 [0171] A program listing of one embodiment of the subject invention is presented in the Appendix hereto.

[0172] Having now described a few embodiments of the invention, and some modifications and variations thereto, it should be apparent to those skilled in the art that the foregoing is merely illustrative and not limiting, having been presented by way of example only. Numerous modifications and other embodiments are within the scope of one of ordinary skill in the art and are contemplated as falling within the scope of the invention as limited only by appended claims and equivalents thereto.

APPENDIX

5

1

Functional Simulator for volume rendering

Copyright, Mitsubishi Electric Information

10

Technology Center America, Inc., 1997, All rights reserved.

15

```
:::::::::::  
cube4/AddressGenerator.C  
:::::::::::  
// AddressGenerator.C  
// (c) Ingmar Bitter '97  
  
// AddressGenerator  
// in : uvw dataset coordinates  
// out: index into corresponding memory module  
  
// Copyright, Mitsubishi Electric Information Technology Center  
// America, Inc., 1997, All rights reserved.  
  
#include "AddressGenerator.h"
```

```
void AddressGenerator::Demo()
```

{

30

```
    AddressGenerator addr;  
    cout << endl << "Demo of class " << typeid(addr).name();  
    cout << endl << "size : " << sizeof(AddressGenerator) << " Bytes";  
    cout << endl << "public member functions:";  
    cout << endl << "AddressGenerator addr; = " << addr;  
    cout << endl << "End of demo of class " << typeid(addr).name() << endl;  
} // Demo
```

```
///////////////////////////////  
// constructors & destructors
```

40

```
// static first init
```

```
int AddressGenerator::numOfChips          = 0;  
int AddressGenerator::numOfPipelinesPerChip = 0;  
int AddressGenerator::blockSize           = 0;  
int AddressGenerator::blockEdge           = 0;  
45   int AddressGenerator::blockFace          = 0;  
int AddressGenerator::blockVolume         = 0;
```

```
AddressGenerator::AddressGenerator()
```

{

50

```
    results.address = new int [numOfPipelinesPerChip];  
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];  
    results.perPipelineControlFlags = new PerPipelineControlFlags  
[numOfPipelinesPerChip];
```

55

```

2
} // constructor

5   AddressGenerator::~AddressGenerator()
{
    if (results.address) { delete results.address; results.address=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
10  if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

15
///////////////////////////////
// show/set data & data properties
//
// - local show/set functions

20 ostream & AddressGenerator::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "blockEdge" = "<<blockFace";
25    os << endl << "blockFace" = "<<blockFace";
    os << endl << "blockVolume" = "<<blockVolume";
    os << endl << "datasetSizeUVW" = "<<datasetSizeUVW";
    os << endl << "rowOfBlocks" = "<<rowOfBlocks";
    os << endl << "sliceOfBlocks" = "<<sliceOfBlocks";
30    os << endl << "U_step" = "<<U_step";
    os << endl << "V_step" = "<<V_step";
    os << endl << "W_step" = "<<W_step";
    os << endl << "u_step" = "<<u_step";
    os << endl << "v_step" = "<<v_step";
    os << endl << "w_step" = "<<w_step";

35    // return complete os
    return os;
}

} // Ostream

40 void AddressGenerator::GlobalSetup(const int setNumOfChips,
                                     const int setNumOfPipelinesPerChip,
                                     const int setBlockSize)
45  {
    numOfChips           = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize            = setBlockSize;
}

```

50

55

```

      3
    blockEdge = blockSize;
    blockFace = blockEdge * blockEdge;
    blockVolume = blockFace * blockEdge;
5   } // GlobalSetup

void AddressGenerator::PerFrameSetup(const Vector3D<int> & setDatasetSizeUVW,
10          const Matrix4x4<int> &
setPipelineToDatasetMatrix)
{
    datasetSizeUVW = setDatasetSizeUVW;
    pipelineToDatasetMatrix = setPipelineToDatasetMatrix;

15   rowOfBlocks = blockVolume * datasetSizeUVW.U();
    sliceOfBlocks = rowOfBlocks * (datasetSizeUVW.V()/blockEdge);

    // linear address offset in u,v,w for complete blocks
    U_step = blockVolume;
20   V_step = rowOfBlocks / (blockEdge*numOfChips); // division because of
storage
    W_step = sliceOfBlocks / (blockEdge*numOfChips); // distributed over
numOfChips

25   // linear address offset in u,v,w for voxels inside a block
    u_step = 1;
    v_step = blockEdge;
    w_step = blockFace;

    // reset pipeline registers
30   for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.address[p] = 0;
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

35   // print debug info
    // static bool first(true); if (first) { cout<<this<<endl; first =
false; }
} // PerFrameSetup

40   /////////////////////////////////
// local computation functions

45   void AddressGenerator::RunForOneClockCycle()
{
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.address[p] = MemoryAddress(inputs.voxelPosXYZ[p].X(),
50                               inputs.voxelPosXYZ[p].Y(),

```

55

```

      inputs.voxelPosXYZ[p].z());
5    results.weightsXYZ[p] = inputs.weightsXYZ[p];
    results.perPipelineControlFlags[p] =
inputs.perPipelineControlFlags[p];
    }
    results.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle
10

int AddressGenerator::ChipIndex (const int u, const int v, const int w)
{
    return ( u / blockEdge +
15          v / blockEdge +
                      w / blockEdge ) % numOfChips;
} // ChipIndex

int AddressGenerator::BlockStartAddress (const int u, const int v, const int w)
20
{
    return ( u / (blockEdge*numOfChips) * U_step +
                      v / blockEdge           * V_step +
                      w / blockEdge           * W_step );
} // BlockStartAddress

25
int AddressGenerator::VoxelBlockOffset (const int u, const int v, const int w)
{
    return ( u % blockEdge * u_step +
                      v % blockEdge * v_step +
                      w % blockEdge * w_step );
30 } // VoxelBlockOffset

int AddressGenerator::MemoryAddress (const int x, const int y, const int z)
{
35    int u,v,w;
    Vector3D<int> posUVW, posXYZ;

    posXYZ(x,y,z);
    posUVW = pipelineToDatasetMatrix * posXYZ;

40    u = posUVW.U();  v = posUVW.V();  w = posUVW.W();

    return BlockStartAddress(u,v,w) + VoxelBlockOffset(u,v,w);
} // MemoryAddress

45 // end of AddressGenerator.C
::::::::::
cube4/AddressGenerator.h
::::::::::
// AddressGenerator.h
50

```

```

// (c) Ingmar Bitter '97

5   // AddressGenerator
// in : uvw dataset coordinates
// out: index into corresponding memory module

// Copyright, Mitsubishi Electric Information Technology Center
10  // America, Inc., 1997, All rights reserved.

#ifndef _AddressGenerator_h_ // prevent multiple includes
#define _AddressGenerator_h_

15  #include "Object.h"
#include "Vector3D.h"
#include "Matrix4x4.h"
#include "Control.h"
#include "FixPointNumber.h"

20  class AddressGeneratorInputs {
public: // pointers
    Vector3D<int> *voxelPosXYZ;
    Vector3D<FixPointNumber> *weightsXYZ;
25  PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

30  class AddressGeneratorResults {
public: // arrays
    int *address;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

35  class AddressGenerator : virtual public Object {
public:

40  static void Demo ();

        // constructors & destructors
        AddressGenerator();
        ~AddressGenerator();
45  // show/set data & data properties
        virtual ostream & Ostream (ostream & ) const;

        static void GlobalSetup (const int setNumOfChips,
50  const int setNumOfPipelinesPerChip,
        const int setBlockSize);

```

```

6
virtual void PerFrameSetup(const Vector3D<int> & setDatasetSizeUVW,
5
    const Matrix4x4<int> & setPipelineToDatasetMatrix);

// local computation functions
    virtual void RunForOneClockCycle();
    virtual /*inline*/ int ChipIndex      (const int u, const int v, const
int w);
10   virtual /*inline*/ int BlockStartAddress (const int u, const int v, const
int w);
    virtual /*inline*/ int VoxelBlockOffset  (const int u, const int v, const
int w);
    virtual /*inline*/ int MemoryAddress     (const int x, const int y, const
int z);
15
public:
    AddressGeneratorInputs inputs;
    AddressGeneratorResults results;

protected:
20   Matrix4x4<int> pipelineToDatasetMatrix;
    static int numOfChips, numOfPipelinesPerChip, blockSize;
    static int blockEdge, blockFace, blockVolume;
    int rowOfBlocks, sliceOfBlocks;
    int U_step, V_step, W_step;
25   int u_step, v_step, w_step;
    Vector3D<int> datasetSizeUVW;
    int lineExtend; // this many addresses are needed per line
    int sliceExtend; // this many addresses are needed per slice
};

30 #endif      // _AddressGenerator_h_
:::::::::::
cube4/ColorLUT.C
:::::::::::
// ColorLUT.C
// (c) Ingmar Bitter '97 / Urs Kanus '97
35

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

40 #include "Cube4.h"
#include "ColorLUT.h"

///////////////////////////////
// Construction/Destruction
///////////////////////////////

45 ColorLUT::ColorLUT()
{
    pLookupTable = 0;
}

```

50

55

```

ColorLUT::~ColorLUT()
{
5
}

void ColorLUT::WriteTable(const char * fileName)
{
    cout << fileName;
10
}

void ColorLUT::ReadTable (const char * fileName)
{
15
    int index;
    int red, green ,blue, alpha;

    if (pLookupTable) delete [] pLookupTable;

    pLookupTable = new LUT[256];
20
    //cout << endl << "lookup Table \\"<< fileName << "\\n";
    ifstream lutFile(fileName);

25
    lutFile >> index;
    while (index <255){
        lutFile >> alpha;
        lutFile >> red;
        lutFile >> green;
50
        lutFile >> blue;
        pLookupTable[index].red    = red;
        pLookupTable[index].green = green;
        pLookupTable[index].blue   = blue;
        pLookupTable[index].alpha = alpha;
        lutFile >> index;
35
    }
}

///////////////////////////////
40 // show/set data & data properties
//



ostream & ColorLUT::Ostream(ostream & os) const
{
45
    // append Light info to os
    os << "not there yet, sorry\n";

    // return complete os
    return os;
50
} // Ostream

```

```

FixPointNumber ColorLUT::Red(int index) const { return pLookupTable[index].red;
5
}
FixPointNumber ColorLUT::Green(int index) const { return
pLookupTable[index].green; }

FixPointNumber ColorLUT::Blue(int index) const { return
10 pLookupTable[index].blue; }

FixPointNumber ColorLUT::Alpha(int index) const { return
pLookupTable[index].alpha; }

FixPointNumber ColorLUT::Kd(int index) const { return pLookupTable[index].kd; }
15
FixPointNumber ColorLUT::Ks(int index) const { return pLookupTable[index].ks; }

void ColorLUT::ComputeLUT(int param) const { cout << param; }
::::::::::
cube4/ColorLUT.h
20 :::::::::::
// ColorLUT.h
// (c) Ingmar Bitter '97 / Urs Kanus '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.
25

#ifndef _ColorLUT_h_ // prevent multiple includes
#define _ColorLUT_h_

30 #include "Misc.h"
#include "Object.h"
#include "FixPointNumber.h"

35 struct LUT {
    FixPointNumber red;
    FixPointNumber green;
    FixPointNumber blue;
    FixPointNumber alpha;
    FixPointNumber kd;
    FixPointNumber ks;
};

40 class ColorLUT : virtual public Object {
public:

    static void Demo ();

    // constructors & destructors
45    ColorLUT ();
    ~ColorLUT ();

    // show/set data & data properties

```

50

55

```

9
// - class Object requirements
virtual ostream & Ostream (ostream & ) const;
5
// - local show/set functions

virtual FixPointNumber Red(int index) const;
virtual FixPointNumber Green(int index) const;
10
virtual FixPointNumber Blue(int index) const;
virtual FixPointNumber Alpha(int index) const;
virtual FixPointNumber Kd(int index) const;
15
virtual FixPointNumber Ks(int index) const;
virtual void ComputeLUT(int param) const;
virtual void WriteTable(const char * fileName);
20
virtual void ReadTable (const char * fileName);

protected:
25
    LUT* pLookupTable;
};

#endif      // _ColorLUT_h_
30
cube4/ComposBufferPipeline.C
::::::::::
// ComposBufferPipeline.C
// (c) Ingmar Bitter '97

35
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposBufferPipeline.h"

void ComposBufferPipeline::Demo()
40
{
    ComposBufferPipeline composBuffer;
    cout << endl << "Demo of class " << typeid(composBuffer).name();
    cout << endl << "size : " << sizeof(ComposBufferPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposBufferPipeline composBuffer; = " << composBuffer;
    cout << endl << "End of demo of class " << typeid(composBuffer).name() <<
45
endl;
} // Demo

```

50

55

10

```

////////// // constructors & destructors
5   // static first init
    int ComposBufferPipeline::numOfChips      = 0;
    int ComposBufferPipeline::numOfPipelinesPerChip = 0;
    int ComposBufferPipeline::blockSize        = 0;
10   Cube4 *ComposBufferPipeline::cube4       = 0;

    ComposBufferPipeline::ComposBufferPipeline()
    {
    } // constructor

15
    ComposBufferPipeline::~ComposBufferPipeline()
    {
    } // destructor

20
////////// // show/set data & data properties

25   ostream & ComposBufferPipeline::Ostream(ostream & os) const
    {
        // append ComposBufferPipeline info to os
        os << typeid(*this).name() << "@" << (void *) this;
        os << endl << " numOfChips           = " << numOfChips;
        os << endl << " numOfPipelinesPerChip = " << numOfPipelinesPerChip;
        os << endl << " chipIndex            = " << chipIndex;
30   os << endl << " blockSliceCoxelFiFo.Size() = "
        " << blockSliceCoxelFiFo.Size();
        os << endl << " volumeSliceCoxelFiFo.Size() = "
        " << volumeSliceCoxelFiFo.Size();

35   // return complete os
        return os;
    } // Ostream

40
////////// // show/set data & data properties
// 
// - local show/set functions

45   void ComposBufferPipeline::GlobalSetup(const int setNumOfChips,
                                           
                                            const int setNumOfPipelinesPerChip,
                                           
                                            const int setBlockSize)
50

```

55

```

11
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
} // GlobalSetup

10 void ComposBufferPipeline::LocalSetup(const int setChipIndex,
{
    const int setPipelineIndex,
    ComposBufferStage & composBufferStage)
15 {
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.coxel = &(composBufferStage.inputs.coxel[pipelineIndex]);
    inputs.perChipControlFlags
        = composBufferStage.inputs.perChipControlFlags;
20
    results.coxel = &(composBufferStage.results.coxel[pipelineIndex]);

    cube4 = composBufferStage(cube4);
} // LocalSetup

25 void ComposBufferPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in SliceVoxelFiFoStage

    // resize fifo's according to dataset size
    // step delay for a z-step within a block
    blockSliceDelay = (cube4->datasetSizeXYZ.X()*blockSize
    ) /
30 (numOfChips*numOfPipelinesPerChip);

    // step delay for a z-step between blocks
    volumeSliceDelay = (cube4->datasetSizeXYZ.X()*cube4->datasetSizeXYZ.Y()
    ) /
35 (numOfChips*numOfPipelinesPerChip);

    blockSliceCoxelFiFo.SetSize(blockSliceDelay);
    volumeSliceCoxelFiFo.SetSize(volumeSliceDelay);

    *(inputs.coxel) = cube4->backgroundCoxel;
    blockSliceCoxelFiFo.Preset(*(inputs.coxel));
40
    volumeSliceCoxelFiFo.Preset(*(inputs.coxel));

    readBigFiFoCounter = readSmallFiFoCounter =
        writeBigFiFoCounter = writeSmallFiFoCounter = -1;
45
} // PerFrameSetup
50

```

55

```

///////////
5 // local computation functions

void ComposBufferPipeline::RunForOneClockCycle()
{
    // reset counters
    if (cube4->composBuff[0].inputs.perChipControlFlags->volumeStart ||
10      inputs.perChipControlFlags->volumeStart ||
        ((readBigFiFoCounter == 0) &&
         (readSmallFiFoCounter == 0) &&
         (writeBigFiFoCounter == 0) &&
         (writeSmallFiFoCounter == 0))) {

15    readBigFiFoCounter = blockSliceDelay;
    readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;

    writeBigFiFoCounter = readBigFiFoCounter;
    writeSmallFiFoCounter = readSmallFiFoCounter;
20
}

///////////
// first read from FiFos into results register

25 // at start of block read from big FiFo
if (readBigFiFoCounter > 0) {
    volumeSliceCoxelFiFo.Read(*(results.coxel));
    --readBigFiFoCounter;
    // if (this == &cube4->sliceCoxelFiFo1[0].sliceCoxelFiFoPipeline[0])
    cout<<"R"<<*(results.coxel)<<volumeSliceCoxelFiFo<<endl;
30
}

35 // in middle and at end of block read from small FiFo
else if (readSmallFiFoCounter > 0) {
    blockSliceCoxelFiFo.Read(*(results.coxel));
    --readSmallFiFoCounter;
    //if (pipelineIndex == 0)      cout<<"r"<<*(results.coxel);
}

///////////
40 // now write to FiFos from inputs register

// at start and in middle of block write to small FiFo
if (writeSmallFiFoCounter > 0) {
    blockSliceCoxelFiFo.Write( *(inputs.coxel) );
    --writeSmallFiFoCounter;
    // if (this == &cube4->sliceCoxelFiFo0[0].sliceCoxelFiFoPipeline[0])
45    cout<<"w";
}

// at end of block write to big FiFo of next chip

```

50

55

```

      13
else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
    cube4->composBuff[c].composBufferPipeline[p].
        volumeSliceCoxelFiFo.Write(*(inputs.coxel) );
    --writeBigFiFoCounter;
    //if (this == &cube4->sliceCoxelFiFo1[0].sliceCoxelFiFoPipeline[0])
    cout<<"W";
}
} // RunForOneClockCycle

////////////////////////////////////////////////////////////////
// internal utility functions
15

// end of ComposBufferPipeline.C
:::::::::::
cube4/ComposBufferPipeline.h
:::::::::::
20 // ComposBufferPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

25 #ifndef _ComposBufferPipeline_h_      // prevent multiple includes
#define _ComposBufferPipeline_h_

30 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Coxel.h"
#include "Control.h"
#include "FiFo.h"

35 class ComposBufferStage;
class Cube4;

40 class ComposBufferPipelineInputs {
public: // pointers
    Coxel *coixel;
    PerChipControlFlags *perChipControlFlags;
};

45 class ComposBufferPipelineResults {
public: // pointers
    Coxel *coixel;
};

50 class ComposBufferPipeline : virtual public Object {

```

55

```

14
public:
5     static void      Demo ();
         // constructors & destructors
         ComposBufferPipeline ();
         ~ComposBufferPipeline ();

10    // show/set data & data properties
         // - class Object requirements
         virtual ostream & Ostream (ostream &) const;
         // - local show/set functions
15    static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           const int setBlockSize);
         virtual void LocalSetup (const int setChipIndex,
20
                           const int setPipelineIndex,
                           ComposBufferStage & composBufferStage);
         virtual void PerFrameSetup ();
         // local computation functions
25
         virtual void RunForOneClockCycle();

public:
30    ComposBufferPipelineInputs inputs;
         ComposBufferPipelineResults results;

protected:
35    Fifo<Coxel> blockSliceCoxelFifo;
         Fifo<Coxel> volumeSliceCoxelFifo;
         int blockSliceDelay, volumeSliceDelay;
         int readSmallFifoCounter, readBigFifoCounter;
         int writeSmallFifoCounter, writeBigFifoCounter;

40    static int   numOfChips, numOfPipelinesPerChip, blockSize;
         static Cube4 *cube4;
         int          chipIndex, pipelineIndex; // only for debugging purpose
};

45    #include "ComposBufferStage.h"
    #include "Cube4.h"
#endif // _ComposBufferPipeline_h_
::::::::::
cube4/ComposBufferStage.C
::::::::::
// ComposBufferStage.C
50

```

15

```

// (c) Ingmar Bitter '97

5 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposBufferStage.h"

void ComposBufferStage::Demo()
{
    ComposBufferStage composBuffer;
    cout << endl << "Demo of class " << typeid(composBuffer).name();
    cout << endl << "size : " << sizeof(ComposBufferStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposBufferStage composBuffer; = " << composBuffer;
    cout << endl << "End of demo of class " << typeid(composBuffer).name() <<
15 endl;
} // Demo

20 /////////////////////////////////
// constructors & destructors

// static first init
int ComposBufferStage::numOfChips          = 0;
int ComposBufferStage::numOfPipelinesPerChip = 0;
25 Cube4 *ComposBufferStage::cube4          = 0;

ComposBufferStage::ComposBufferStage() : composBufferPipeline(0)
{
    composBufferPipeline = new ComposBufferPipeline [numOfPipelinesPerChip];
    results.coxel = new Coxel [numOfPipelinesPerChip + 1];
30 } // constructor

ComposBufferStage::~ComposBufferStage()
{
    if (composBufferPipeline) { delete composBufferPipeline;
35 composBufferPipeline=0; }
    if (results.coxel) { delete results.coxel; results.coxel=0; }
} // destructor

40 /////////////////////////////////
// show/set data & data properties

ostream & ComposBufferStage::Ostream(ostream & os) const
{
    // append ComposBufferStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips           = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;
50

```

55

```

    // return complete os
5   return os;

} // Ostream

10  ///////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

15  void ComposBufferStage::GlobalSetup(const int setNumOfChips,
{
    const int setNumOfPipelinesPerChip,
    const int setBlockSize,
    Cube4 *setCube4)

20  {
    numOfChips      = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    cube4          = setCube4;
    ComposBufferPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
25
        blockSize);
} // GlobalSetup

30  void ComposBufferStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composBufferPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup

35

40  void ComposBufferStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.coxel[p] = cube4->backgroundCoxel;
    }

45  for (p=0; p<numOfPipelinesPerChip; ++p) {
        composBufferPipeline[p].PerFrameSetup();
    }

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
50
}

```

```

} // PerFrameSetup

5
///////////////////////////////
// local computation functions

void ComposBufferStage::RunForOneClockCycle()
{
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composBufferPipeline[p].RunForOneClockCycle();
    }
} // RunForOneClockCycle

15
///////////////////////////////
// internal utility functions

20 // end of ComposBufferStage.C
:::::::::::
cube4/ComposBufferStage.h
:::::::::::
// ComposBufferStage.h
// (c) Ingmar Bitter '97
25

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30 #ifndef _ComposBufferStage_h_ // prevent multiple includes
#define _ComposBufferStage_h_

35 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Coxel.h"
#include "ComposBufferPipeline.h"
#include "Cube4.h"

40 class ComposBufferStageInputs {
public:
    Coxel *coixel; // pointer
    PerChipControlFlags *perChipControlFlags;
};

45 class ComposBufferStageResults {
public:
    Coxel *coixel; // array
};

50 class ComposBufferStage : virtual public Object {
public:

```

```

static void      Demo ();
5
    // constructors & destructors
    ComposBufferStage ();
    ~ComposBufferStage ();

10   // show/set data & data properties
    // - class Object requirements
    virtual ostream & Ostream (ostream & ) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
15
        const int setNumOfPipelinesPerChip,
        const int setBlockSize,
        Cube4 *setCube4);

        virtual void LocalSetup (const int setChipIndex);
        virtual void PerFrameSetup();
        // local computation functions
        virtual void RunForOneClockCycle();

25
public:
    ComposBufferPipeline *composBufferPipeline;
    ComposBufferStageInputs inputs;
    ComposBufferStageResults results;
30
protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    int          chipIndex; // only for debugging purpose
35
    friend class ComposBufferPipeline;
};

#ifndef      // _ComposBufferStage_h_
40
:::cube4/ComposLinXPipeline.C
::::::::::
// ComposLinXPipeline.C
// (c) Ingmar Bitter '97
45
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposLinXPipeline.h"

50
void ComposLinXPipeline::Demo()
{
    ComposLinXPipeline composLinX;

```

```

19
cout << endl << "Demo of class " << typeid(composLinX).name();
cout << endl << "size : " << sizeof(ComposLinXPipeline) << " Bytes";
cout << endl << "public member functions:";
cout << endl << "ComposLinXPipeline composLinX; = " << composLinX;
cout << endl << "End of demo of class " << typeid(composLinX).name() <<
endl;
} // Demo

10
///////////////////////////////
// constructors & destructors

11
// static first init
12 int ComposLinXPipeline::numOfChips          = 0;
13 int ComposLinXPipeline::numOfPipelinesPerChip = 0;
14 int ComposLinXPipeline::blockSize           = 0;
15 Cube4 *ComposLinXPipeline::cube4            = 0;

16
17 ComposLinXPipeline::ComposLinXPipeline()
18 {
19     // Number of clock cycles to process a partial beam inside a block
20     int partialBeamDelay = blockSize / numOfPipelinesPerChip;
21
22     // Delay for communication: 16 bits, 4 bits / cycle, double frequency => 2
23     // cycles
24     int communicationDelay = 2;
25
26     partialBeamCoxelFiFo.SetSize(partialBeamDelay);
27     partialBeamWeightsFiFo.SetSize(partialBeamDelay);
28     partialBeamPerPipelineControlFlagsFiFo.SetSize(partialBeamDelay);
29     partialBeamPerChipControlFlagsFiFo.SetSize(partialBeamDelay);
30
31     communicationDelayCoxelFiFo.SetSize(communicationDelay);
32     communicationDelayWeightsFiFo.SetSize(communicationDelay);
33     communicationDelayPerPipelineControlFlagsFiFo.SetSize(communicationDelay);
34     communicationDelayPerChipControlFlagsFiFo.SetSize(communicationDelay);
35 } // constructor

36
37 ComposLinXPipeline::~ComposLinXPipeline()
38 {
39 } // destructor

40
///////////////////////////////
// show/set data & data properties

41
42 ostream & ComposLinXPipeline::Ostream(ostream & os) const
43 {
44     // append ComposLinXPipeline info to os
45     os << typeid(*this).name() << "@" << (void *) this;
46     os << endl << "    numOfChips      = " << numOfChips;
47
48
49
50
51
52
53
54
55

```

```

20
    os << endl << "numOfPipelinesPerChip";
    os << endl << "chipIndex" = " << chipIndex;
5

// return complete os
return os;

} // Ostream
10

///////////////////////////////
// show/set data & data properties
//
// - local show/set functions
15

void ComposLinXPipeline::GlobalSetup(const int setNumOfChips,
                                      const int setNumOfPipelinesPerChip,
20
                                      const int setBlockSize)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
} // GlobalSetup
25

void ComposLinXPipeline::LocalSetup(const int setChipIndex,
                                     const int setPipelineIndex,
                                     ComposLinXStage & composLinXStage)
30
{
    chipIndex = setChipIndex;
    pipelineIndex(setPipelineIndex, numOfPipelinesPerChip+1);

    inputs.coxelLeft   = &(composLinXStage.computation.coxel[pipelineIndex-1]);
    inputs.coxelMiddle = &(composLinXStage.computation.coxel[pipelineIndex]);
    inputs.weightsXYZ =
&(composLinXStage.computation.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = &composLinXStage.computation.perChipControlFlags;
    inputs.perPipelineControlFlags
40
        =
&(composLinXStage.computation.perPipelineControlFlags[pipelineIndex]);

    results.coxel = &(composLinXStage.results.coxel[pipelineIndex]);
    results.weightsXYZ = &(composLinXStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
45
        = &(composLinXStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(composLinXStage.results.perChipControlFlags);
}

```

50

55

```

      21
    cube4 = composLinXStage(cube4);
} // LocalSetup
5

void ComposLinXPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in ComposLinXStage
    partialBeamCoxelFiFo.Preset(*(inputs.coxelMiddle));
10   partialBeamWeightsFiFo.Preset(*(inputs.weightsXYZ));

    partialBeamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags))
;    partialBeamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

15   communicationDelayCoxelFiFo.Preset(*(inputs.coxelMiddle));
    communicationDelayWeightsFiFo.Preset(*(inputs.weightsXYZ));

    communicationDelayPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControl
Flags));
20   communicationDelayPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));
} // PerFrameSetup

///////////////////////////////
25   // local computation functions

void ComposLinXPipeline::RunForOneClockCycle()
{
    if (cube4->cubeMode == Cube4Classic) {

        // complete selection from ComposSelX

        if (inputs.perChipControlFlags->xStep == Left) {
            // data comes form the right (which happens in ComposSelX)
            // L      M      R
            // o      o<----o
35       *(results.coxel) = *(inputs.coxelMiddle);
        }
        else if (inputs.perChipControlFlags->xStep == Right) {
            // data comes from the left
            // L      M      R
            // o---->o      o
40       *(results.coxel) = *(inputs.coxelLeft);
        }
        else if (inputs.perChipControlFlags->xStep == Middle) {
            // data goes straight
            // L      M      R
            // o      o      o
45       *(results.coxel) = *(inputs.coxelMiddle);
        }
        else { ERROR( "Wrong xstep in ComposLinXPipeline!" ); }
    }
}
50

```

55

```

5      else if (cube4->cubeMode == Cube4Light) {

10     // interpolating between coixels,
11     // interpolation weights depending on x weight

15     FixPointNumber interpolationWeight;

20     if (inputs.weightsXYZ->X() < 0) {
21       // weight pointing left, starting at middle coixel
22       // L      M
23       // o <-o      // dx < 0
24       interpolationWeight = 1.0 + inputs.weightsXYZ->X();
25     }
26     else {
27       // weight pointing right, starting at left coixel
28       // L      M
29       // o-> o      // dx > 0
30       interpolationWeight = inputs.weightsXYZ->X();
31     }

35     /////////////////////////////////
36     // Linear interpolation
37     // c = w(b-a)+a
38     //
39     // w=0 -> c=a
40     // w=1 -> c=b

45     a = *inputs.coxelLeft;
46     b = *inputs.coxelMiddle;

50     // if current pipeline is working on a starting ray
51     // set out of volume samples to background color
52     if (inputs.perPipelineControlFlags->startOfRay) {
53       if (inputs.weightsXYZ->X() < 0) a = cube4->backgroundCoxel;
54       else                                b = cube4->backgroundCoxel;
55     }

56     results.coxel->r = interpolationWeight * (b.r - a.r) + a.r;
57     results.coxel->g = interpolationWeight * (b.g - a.g) + a.g;
58     results.coxel->b = interpolationWeight * (b.b - a.b) + a.b;
59     results.coxel->a = interpolationWeight * (b.a - a.a) + a.a;
60   }

65 } // RunForOneClockCycle

70 /////////////////////////////////
71 // internal utility functions

75 // end of ComposLinXPipeline.C
76 ::::::::::::::::::::

```

23

```

cube4/ComposLinXPipeline.h
::::::::::
// ComposLinXPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _ComposLinXPipeline_h_      // prevent multiple includes
#define _ComposLinXPipeline_h_

#include "Misc.h"
#include "Object.h"
#include "FiFo.h"
#include "Coxel.h"
#include "ModInt.h"
#include "Control.h"
#include "FixPointNumber.h"
#include "Vector3D.h"

typedef Vector3D<FixPointNumber> FixPointVector3D;

class ComposLinXStage;
class Cube4;

class ComposLinXPipelineInputs {
public: // pointers
    Coxel *coixelLeft;
    Coxel *coixelMiddle;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

class ComposLinXPipelineResults {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

class ComposLinXPipeline : virtual public Object {
public:

    static void Demo ();

    // constructors & destructors
    ComposLinXPipeline ();
    ~ComposLinXPipeline ();
};

```

55

```

24
// show/set data & data properties
// - class Object requirements
5    virtual ostream & Ostream (ostream & ) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
10   const int setNumOfPipelinesPerChip,
               const int setBlockSize);

virtual void LocalSetup(const int setChipIndex,
15   const int setPipelineIndex,
               ComposLinXStage & composLinXStage);
virtual void PerFrameSetup();
// local computation functions
               virtual void RunForOneClockCycle();

20 public:
    ComposLinXPipelineInputs inputs;
    ComposLinXPipelineResults results;

protected:
25    Fifo<Coxel>           partialBeamCoxelFifo;
    Fifo<FixPointVector3D>  partialBeamWeightsFifo;
    Fifo<PerPipelineControlFlags> partialBeamPerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags> partialBeamPerChipControlFlagsFifo;

30    Fifo<Coxel>           communicationDelayCoxelFifo;
    Fifo<FixPointVector3D>  communicationDelayWeightsFifo;
    Fifo<PerPipelineControlFlags> communicationDelayPerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags> communicationDelayPerChipControlFlagsFifo;

35    static int   numOfChips, numOfPipelinesPerChip, blockSize;
    static Cube4 *cube4;
    int         chipIndex;
    ModInt      pipelineIndex;
    Coxel       a,b;

        friend class ComposLinXStage;
40    friend class Cube4;
};

#include "ComposLinXStage.h"
#include "Cube4.h"

45 #endif // _ComposLinXPipeline_h_
:::::::::::
cube4/ComposLinXStage.C
:::::::::::
// ComposLinXStage.C
50 // (c) Ingmar Bitter '97

```

```

// Copyright, Mitsubishi Electric Information Technology Center
5 // America, Inc., 1997, All rights reserved.

#include "ComposLinXStage.h"

void ComposLinXStage::Demo()
{
    ComposLinXStage composLinX;
    cout << endl << "Demo of class " << typeid(composLinX).name();
    cout << endl << "size : " << sizeof(ComposLinXStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposLinXStage composLinX; = " << composLinX;
    cout << endl << "End of demo of class " << typeid(composLinX).name() <<
10 endl;
} // Demo

///////////////////////////////
// constructors & destructors
20

// static first init
int ComposLinXStage::numOfChips           = 0;
int ComposLinXStage::numOfPipelinesPerChip = 0;
Cube4 *ComposLinXStage::cube4 = 0;

25 ComposLinXStage::ComposLinXStage()
{
    composLinXPipeline = new ComposLinXPipeline [numOfPipelinesPerChip];

    computation.coxel
        = new Coxel [numOfPipelinesPerChip+1];
    computation.weightsXYZ
        = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    computation.perPipelineControlFlags
        = new PerPipelineControlFlags [numOfPipelinesPerChip+1]; // +1 just
for debugging
30
    results.coxel
        = new Coxel [numOfPipelinesPerChip];
    results.weightsXYZ
        = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags
        = new PerPipelineControlFlags [numOfPipelinesPerChip];
40
} // defaultconstructor

ComposLinXStage::~ComposLinXStage()
{
45     if (composLinXPipeline) { delete composLinXPipeline; composLinXPipeline=0;
}

    if (computation.coxel) {
50

```

```

26
    delete computation.coxel;
    computation.coxel=0;
5     }
    if (computation.weightsXYZ) {
        delete computation.weightsXYZ;
        computation.weightsXYZ=0;
    }
10    if (computation.perPipelineControlFlags) {
        delete computation.perPipelineControlFlags;
        computation.perPipelineControlFlags=0;
    }

    if (results.coxel) { delete results.coxel; results.coxel=0; }
15    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
    }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
20 } // destructor

///////////////////////////////
// show/set data & data properties

25
ostream & ComposLinXStage::Ostream(ostream & os) const
{
    // append ComposLinXStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips"           = " << numOfChips;
30    os << endl << "  numofPipelinesPerChip" = " << numofPipelinesPerChip;
    os << endl << "  chipIndex"           = " << chipIndex;

    // return complete os
    return os;
35 }
} // Ostream

///////////////////////////////
// show/set data & data properties
40 // - local show/set functions

void ComposLinXStage::GlobalSetup(const int setNumOfChips,
45        const int setNumOfPipelinesPerChip,
        Cube4 *setCube4)
{
    numOfChips           = setNumOfChips;
50

```

```

27
numOfPipelinesPerChip =           setNumOfPipelinesPerChip;
cube4                  = setCube4;
5   ComposLinXPipeline::GlobalSetup(numOfChips,
                                   
                                    numOfPipelinesPerChip,
                                   
                                    cube4->blockSize);
10 } // GlobalSetup

void ComposLinXStage::LocalSetup(const int setChipIndex)
{
    chipIndex(setChipIndex,numOfChips);
15   for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composLinXPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup

20
void ComposLinXStage::PerFrameSetup()
{
    // reset pipeline registers
    int p;
25   readBufferCoxel = 0;
   communicationCoxel = 0;
   for (p=0; p<numOfPipelinesPerChip; ++p) {
       computation.coxel[p] = 0;
       computation.weightsXYZ[p](0,0,0);
       computation.perPipelineControlFlags[p].Reset();
30   }
   computation.coxel[numOfPipelinesPerChip] = 0;

   for (p=0; p<numOfPipelinesPerChip; ++p) {
       results.coxel[p] = cube4->backgroundCoxel;
35   results.weightsXYZ[p](0,0,0);
       results.perPipelineControlFlags[p].Reset();
   }
   results.perChipControlFlags.Reset();

40   for (p=0; p<numOfPipelinesPerChip; ++p) {
       composLinXPipeline[p].PerFrameSetup();
   }
} // PerFrameSetup

45   /////////////////////////////////
// local computation functions

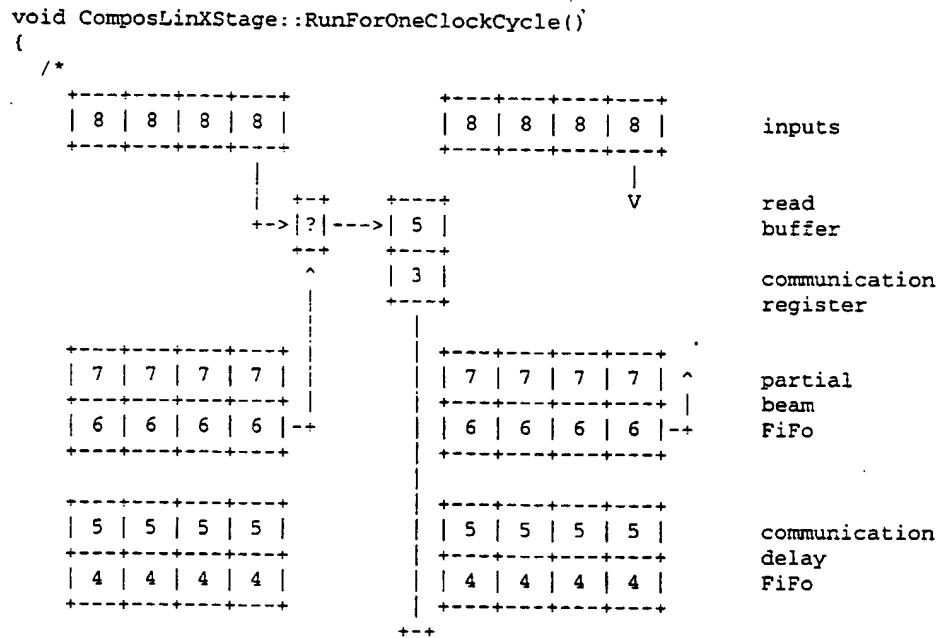
void ComposLinXStage::CommunicateForOneClockCycle()
{
50   ///////////////////////////////
// fill communication register

```

```

28
// only sent data at beginning of a    block
if (inputs.perChipControlFlags->xBlockEnd) {
5
    // write readbuffer to communication register
    cube4->composLinX[chipIndex+1].communicationCoxel =
        cube4->composLinX[chipIndex+1].readBufferCoxel;
    cube4->composLinX[chipIndex+1].communicationPerPipelineControlFlags =
        cube4->composLinX[chipIndex+1].readBufferPerPipelineControlFlags;
10
    // leftmost chip sends data as early as possible
    if (inputs.perChipControlFlags->rightmostChip) {
        cube4->composLinX[chipIndex+1].readBufferCoxel =
            inputs.coxel[0];
    cube4->composLinX[chipIndex+1].readBufferPerPipelineControlFlags =
        inputs.perPipelineControlFlags[0];
15
    }
    // remaining chips send (blockSize/numOfPipelinesPerChip) later
    else {
        cube4->composLinX[chipIndex+1].readBufferCoxel =
            composLinXPipeline[0].partialBeamCoxelFiFo.Peek(0);
        cube4->composLinX[chipIndex+1].readBufferPerPipelineControlFlags =
            composLinXPipeline[0].partialBeamPerPipelineControlFlagsFiFo.Peek(0);
20
    }
}
25
} // CommunicateForOneClockCycle

```




```

30
computation.weightsXYZ[p]
    = composLinXPipeline[p].communicationDelayWeightsFiFo.Read();
5   computation.perPipelineControlFlags[p]
    =
composLinXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Read();
}
computation.perChipControlFlags
    = composLinXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Read();
10

///////////////////////////////
// Write to communication delay Fifo
for (p=0; p<numOfPipelinesPerChip; ++p) {
    composLinXPipeline[p].communicationDelayCoxelFiFo.Write
15   (composLinXPipeline[p].partialBeamCoxelFiFo.Read());
    composLinXPipeline[p].communicationDelayWeightsFiFo.Write
    (composLinXPipeline[p].partialBeamWeightsFiFo.Read());
    composLinXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Write
    (composLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Read());
}
20
composLinXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Write
    (composLinXPipeline[0].partialBeamPerChipControlFlagsFiFo.Read());

/////////////////////////////
// Write to partial beam Fifo
25 for (p=0; p<numOfPipelinesPerChip; ++p) {
    composLinXPipeline[p].partialBeamCoxelFiFo.Write
    (inputs.coxel[p]);
    composLinXPipeline[p].partialBeamWeightsFiFo.Write
    (inputs.weightsXYZ[p]);
    composLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Write
30   (inputs.perPipelineControlFlags[p]);
}
composLinXPipeline[0].partialBeamPerChipControlFlagsFiFo.Write
    (*inputs.perChipControlFlags);

35 } // RunForOneClockCycle

// end of ComposLinXStage.C
:::::::::::
cube4/ComposLinXStage.h
:::::::::::
40 // ComposLinXStage.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45 #ifndef _ComposLinXStage_h_    // prevent multiple includes
#define _ComposLinXStage_h_

#include "Misc.h"
#include "Object.h"
50 #include "Coxel.h"

```

```

5      #include "Control.h"
#include "ComposLinXPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

10     class Cube4;

15     class ComposLinXStageInputs {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

20     class ComposLinXStageResults {
public: // arrays
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

25     class ComposLinXStage : virtual public Object {
public:

30         static void Demo ();

            // constructors & destructors
            ComposLinXStage ();
            ~ComposLinXStage ();

35         // show/set data & data properties
            // - class Object requirements
            virtual ostream & Ostream (ostream &) const;

40         // - local show/set functions
            static void GlobalSetup (const int setNumberOfChips,
                                    const int setNumberOfPipelinesPerChip,
                                    Cube4 *setCube4);
            virtual void LocalSetup (const int setChipIndex);
            virtual void PerFrameSetup ();
// local computation functions
            virtual void CommunicateForOneClockCycle();
            virtual void RunForOneClockCycle();

45         public:
            ComposLinXPipeline *composLinXPipeline;
            ComposLinXStageInputs inputs;

50
55

```

```

32
ComposLinXStageResults computation;
ComposLinXStageResults results;

5 protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;

    ModInt        chipIndex;
    Coxel         readBufferCoxel;
    PerPipelineControlFlags readBufferPerPipelineControlFlags; // only for
debugging
    Coxel         communicationCoxel;
    PerPipelineControlFlags communicationPerPipelineControlFlags; // only for
debugging
15

    friend class ComposLinXPipeline;
    friend class Cube4;
};

20 #endif // _ComposLinXStage_h_
:::::::::::
cube4/ComposLinYPipeline.C
:::::::::::
// ComposLinYPipeline.C
// (c) Ingmar Bitter '97

25 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposLinYPipeline.h"
30 void ComposLinYPipeline::Demo()
{
    ComposLinYPipeline composLinY;
    cout << endl << "Demo of class " << typeid(composLinY).name();
    cout << endl << "size : " << sizeof(ComposLinYPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposLinYPipeline composLinY; = " << composLinY;
    cout << endl << "End of demo of class "<< typeid(composLinY).name() <<
endl;
} // Demo
40

///////////////////////////////
// constructors & destructors

45 // static first init
int ComposLinYPipeline::numOfChips          = 0;
int ComposLinYPipeline::numOfPipelinesPerChip = 0;
int ComposLinYPipeline::blockSize            = 0;
Cube4 *ComposLinYPipeline::cube4             = 0;

50 ComposLinYPipeline::ComposLinYPipeline()

```

```

33
{
} // constructor
5

ComposLinYPipeline::~ComposLinYPipeline()
{
} // destructor
10

////////////////////////////// show/set data & data properties

15    ostream & ComposLinYPipeline::Ostream(ostream & os) const
{
    // append ComposLinYPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "numOfChips" = " << numOfChips;
20    os << endl << "numOfPipelinesPerChip" = " << numOfPipelinesPerChip;
    os << endl << "chipIndex" = " << chipIndex;

    // return complete os
    return os;
25 } // Ostream

////////////////////////////// show/set data & data properties
30 // - local show/set functions

35 void ComposLinYPipeline::GlobalSetup(const int setNumOfChips,
        const int setNumOfPipelinesPerChip,
        const int setBlockSize)
{
    40    numOfChips = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize = setBlockSize;
} // GlobalSetup

45 void ComposLinYPipeline::LocalSetup(const int setChipIndex,
        const int setPipelineIndex,
        ComposLinYStage & composLinYStage)
50 {
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;
}

```

```

5      inputs.coxel = &(composLinYStage.inputs.coxel[pipelineIndex]);
inputs.weightsXYZ = &(composLinYStage.inputs.weightsXYZ[pipelineIndex]);
inputs.perChipControlFlags
    = composLinYStage.inputs.perChipControlFlags;
inputs.perPipelineControlFlags
    = &(composLinYStage.inputs.perPipelineControlFlags[pipelineIndex]);

10     results.coxel = &(composLinYStage.results.coxel[pipelineIndex]);
results.weightsXYZ = &(composLinYStage.results.weightsXYZ[pipelineIndex]);
results.perPipelineControlFlags
    = &(composLinYStage.results.perPipelineControlFlags[pipelineIndex]);
results.perChipControlFlags
    = &(composLinYStage.results.perChipControlFlags);

15     cube4 = composLinYStage(cube4);
} // LocalSetup

20 void ComposLinYPipeline::PerFrameSetup()
{
    // resize fifo's according to dataset size
    // step delay for a y-step within a block
    beamDelay = (cube4->datasetSizeXYZ.X())
        / (numOfChips*numOfPipelinesPerChip);

25     // step delay for a y-step between blocks
    blockBeamDelay = beamDelay * blockSize;

    beamCoxelFiFo.SetSize(beamDelay);
    blockBeamCoxelFiFo.SetSize(blockBeamDelay);

30     beamCoxelFiFo.Preset(*(inputs.coxel));
    blockBeamCoxelFiFo.Preset(*(inputs.coxel));

    readBigFiFoCounter = readSmallFiFoCounter =
        writeBigFiFoCounter = writeSmallFiFoCounter = -1;
} // PerFrameSetup

40 // local computation functions

void ComposLinYPipeline::RunForOneClockCycle()
{
    // reset counters
    if (inputs.perChipControlFlags->volumeStart ||
45        (readBigFiFoCounter == 0) &&
        (readSmallFiFoCounter == 0) &&
        (writeBigFiFoCounter == 0) &&
        (writeSmallFiFoCounter == 0))) {

```

```

      35
      readBigFiFoCounter = beamDelay;
      readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;
5
      writeBigFiFoCounter = readBigFiFoCounter;
      writeSmallFiFoCounter = readSmallFiFoCounter;
    }

10   /////////////////
// first read from Fifos into results register

15   // at start of block read from big Fifo
if (readBigFiFoCounter > 0) {
    blockBeamCoxelFiFo.Read(fifoCoxel);
    --readBigFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0)
cout<<"R"<<*(results.coxel)<<volumeSliceGradientFiFo<<endl;
}

20   // in middle and at end of block read from small Fifo
else if (readSmallFiFoCounter > 0) {
    beamCoxelFiFo.Read(fifoCoxel);
    --readSmallFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0) cout<<"r";
}

25   // control always goes the same way as data with weight 0 (step middle)
// thus control comes from the input registers
*results.weightsXYZ = *inputs.weightsXYZ;
*results.perPipelineControlFlags = *inputs.perPipelineControlFlags;
*results.perChipControlFlags = *inputs.perChipControlFlags;

30   ///////////////
// buffer writing

35   ///////////////
// now write to Fifos from inputs register

40   // at start and in middle of block write to small Fifo
if (writeSmallFiFoCounter > 0) {
    beamCoxelFiFo.Write(*(inputs.coxel));
    --writeSmallFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0) cout<<"w";
}

45   // at end of block write to big Fifo of next chip
else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
    ComposLinYPipeline *next(&cube4-
>composLinY[c].composLinYPipeline[p]);
    next->blockBeamCoxelFiFo.Write(*inputs.coxel);
    --writeBigFiFoCounter;
    //if (chipIndex == 0 && pipelineIndex == 0) cout<<"W";
}

```

```

}
5
///////////////////////////////
// select/interpolate results from Fifo or inputs
// depending on yStep or y-weight

10    if (cube4->cubeMode == Cube4Classic) {

15        if (inputs.perChipControlFlags->yStep == Down) {
            // data comes from the top
            // T      M      B  (data src)
            // D      M      U  (ray step)
            // o-----o
            *(results.coxel) = fifoCoxel;
        }
        else if (inputs.perChipControlFlags->yStep == Up) {
            // data comes from the bottom
            // T      M      B  (data src)
            // D      M      U  (ray step)
            // o      o<----o
            *(results.coxel) = *(inputs.coxel);
        }
20        if (inputs.perChipControlFlags->yStep == Middle) {
            // data goes straight
            // T      M      B  (data src)
            // D      M      U  (ray step)
            // o      o      o
            *(results.coxel) = fifoCoxel;
        }
30        else { ERROR( "Wrong ystep in ComposLinYPipeline!" ); }

35        if (inputs.perPipelineControlFlags->startOfRay) {
            *(results.coxel) = cube4->backgroundCoxel;
        }
    }

    else if (cube4->cubeMode == Cube4Light) {

40        FixPointNumber interpolationWeight;

        if (inputs.weightsXYZ->Y() <= 0) {
            // data comes from the top or goes straight
            // T      M      B  (weight / data src)
            // o-----o
45        interpolationWeight = 1.0 + inputs.weightsXYZ->Y();
        }
        else { // inputs.weightsXYZ->Y() > 0
            // data comes from the bottom
            // T      M      B  (weight / data src)
            // o      o<----o
50        interpolationWeight = inputs.weightsXYZ->Y();
    }
}

```

```

5      ///////////////////////////////////////////////////
// Linear interpolation
// c = w(b-a)+a
//
// w=0 -> c=a
// w=1 -> c=b

10     Coxel a(fifoCoxel);
      Coxel b(*inputs.coxel);

      // if current pipeline is working on a starting ray
      // set out of volume samples to background color
15     if (inputs.perPipelineControlFlags->startOfRay) {
          if (inputs.weightsXYZ->Y() < 0) a = cube4->backgroundCoxel;
          else                                b = cube4->backgroundCoxel;
      }

20     results.coxel->r = interpolationWeight * (b.r - a.r) + a.r;
     results.coxel->g = interpolationWeight * (b.g - a.g) + a.g;
     results.coxel->b = interpolationWeight * (b.b - a.b) + a.b;
     results.coxel->a = interpolationWeight * (b.a - a.a) + a.a;
}
} // RunForOneClockCycle

25

///////////////////////////////////////////////////
// internal utility functions

30     // end of ComposLinYPipeline.C
:::::::::::
cube4/ComposLinYPipeline.h
:::::::::::
// ComposLinYPipeline.h
// (c) Ingmar Bitter '97
35

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

40     #ifndef _ComposLinYPipeline_h_           // prevent multiple includes
#define _ComposLinYPipeline_h_

45     #include "Misc.h"
     #include "Object.h"
     #include "FiFo.h"
     #include "Coxel.h"
     #include "Control.h"
     #include "FixPointNumber.h"

50     typedef Vector3D<FixPointNumber> FixPointVector3D;
     class ComposLinYStage;

```

38

```

class Cube4;

5   class ComposLinYPipelineInputs {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
10   PerPipelineControlFlags *perPipelineControlFlags;
};

15  class ComposLinYPipelineResults {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
20};

25  class ComposLinYPipeline : virtual public Object {
public:

25    static void      Demo ();

        // constructors & destructors
        ComposLinYPipeline ();
        ~ComposLinYPipeline ();

30    // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream &) const;

35    // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,
                                const int setNumOfPipelinesPerChip,
                                const int setBlockSize);

40    virtual void LocalSetup(const int setChipIndex,
                            const int setPipelineIndex,
                            ComposLinYStage & composLinYStage);
45    virtual void PerFrameSetup();
    // local computation functions
    virtual void RunForOneClockCycle();

50  public:
        ComposLinYPipelineInputs  inputs;
        ComposLinYPipelineResults results;

```

55

```

protected:
    Fifo<Coxel> beamCoxelFifo;
    Fifo<Coxel> blockBeamCoxelFifo;

5      Coxel fifoCoxel;

        int beamDelay, blockBeamDelay;

10     int readSmallFiFoCounter, readBigFiFoCounter;
        int writeSmallFiFoCounter, writeBigFiFoCounter;

        static int numOfChips, numOfPipelinesPerChip, blockSize;
        static Cube4 *cube4;
        int chipIndex, pipelineIndex;

15     friend class ComposLinYStage;
        friend class Cube4;
};

20 #include "ComposLinYStage.h"
#include "Cube4.h"

#define _ComposLinYPipeline_h_
::::::::::
cube4/ComposLinYStage.C
::::::::::
// ComposLinYStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30 #include "ComposLinYStage.h"

void ComposLinYStage::Demo()
{
    ComposLinYStage composLinY;
    cout << endl << "Demo of class " << typeid(composLinY).name();
    cout << endl << "size : " << sizeof(ComposLinYStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposLinYStage composLinY; = " << composLinY;
    cout << endl << "End of demo of class " << typeid(composLinY).name()

40 << endl;
} // Demo

///////////////////////////////
// constructors & destructors
45 // static first init
int ComposLinYStage::numOfChips          = 0;
int ComposLinYStage::numOfPipelinesPerChip = 0;
Cube4 *ComposLinYStage::cube4 = 0;

```

50

55

41

```

    const int setBlockSize,
5      Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    cube4               = setCube4;
10     ComposLinYPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
                                         setBlockSize);
} // GlobalSetup

15 void ComposLinYStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composLinYPipeline[p].LocalSetup(chipIndex,p,*this);
    }
20 } // LocalSetup

void ComposLinYStage::PerFrameSetup()
{
25     int p;
     // reset pipeline registers
     for (p=0; p<numOfPipelinesPerChip; ++p) {
         results.coxel[p] = cube4->backgroundCoxel;
         results.weightsXYZ[p](0,0,0);
30         results.perPipelineControlFlags[p].Reset();
     }
     results.perChipControlFlags.Reset();

     for (p=0; p<numOfPipelinesPerChip; ++p) {
         composLinYPipeline[p].PerFrameSetup();
35     }

     // print debug info
     //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

40 //////////////////////////////////////////////////////////////////
// local computation functions

void ComposLinYStage::RunForOneClockCycle()
45 {
    // communication

    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composLinYPipeline[p].RunForOneClockCycle();
50
}

```

55

```

        }
    } // RunForOneClockCycle
5

///////////////////////////////
// internal utility functions

10   // end of ComposLinYStage.C
:::::::
cube4/ComposLinYStage.h
:::::::
// ComposLinYStage.h
15   // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

20   #ifndef _ComposLinYStage_h_    // prevent multiple includes
#define _ComposLinYStage_h_

25   #include "Misc.h"
#include "Object.h"
#include "Vector3D.h"
#include "Coxel.h"
#include "Control.h"
#include "ComposLinYPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"
30

35   class ComposLinYStageInputs {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags    *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

40   class ComposLinYStageResults {
public: // arrays
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags    perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
45 };

50   class ComposLinYStage : virtual public Object {
public:
    static void      Demo ();

```

55

```

43
// constructors & destructors
ComposLinYStage ();
~ComposLinYStage ();

5
// show/set data & data properties
// - class Object requirements
virtual ostream & Ostream (ostream & ) const;

10
// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
                        const int setNumOfPipelinesPerChip,
                        const int setBlockSize,
                        Cube4 *setCube4);
virtual void LocalSetup (const int setChipIndex);
virtual void PerFrameSetup ();

20
// local computation functions
virtual void RunForOneClockCycle();

public:
    ComposLinYPipeline *composLinYPipeline;
    ComposLinYStageInputs inputs;
    ComposLinYStageResults results;

25
protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    int          chipIndex; // only for debugging purpose

30
        friend class ComposLinYPipeline;
};

35
#endif // _ComposLinYStage_h_
::::::::::
cube4/ComposPipeline.C
::::::::::
// ComposPipeline.C
// (c) Ingmar Bitter '97

40
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

45
// Premultiplied Colors (C implies A*C)
//
// Back-to-Front
// Cacc = (1-Anew)Cacc + Cnew
// Aacc = (1-Anew)Aacc + Anew
//
50
// Front-to-Back
// Cacc = (1-Aacc)Cnew + Cacc
// Aacc = (1-Aacc)Anew + Aacc

```

55

```

#include "ComposPipeline.h"

5   void ComposPipeline::Demo()
{
    ComposPipeline compos;
    cout << endl << "Demo of class " << typeid(compos).name();
    cout << endl << "size : " << sizeof(ComposPipeline) << " Bytes";
10  cout << endl << "public member functions:";
    cout << endl << "ComposPipeline compos; = " << compos;
    cout << endl << "End of demo of class " << typeid(compos).name() << endl;
} // Demo

15  /////////////////////////////////
// constructors & destructors

// static first init
int ComposPipeline::numOfChips           = 0;
20  int ComposPipeline::numOfPipelinesPerChip = 0;
Cube4 *ComposPipeline::cube4            = 0;

ComposPipeline::ComposPipeline()
{
25  } // constructor

ComposPipeline::~ComposPipeline()
{
} // destructor

30  /////////////////////////////////
// show/set data & data properties

35  ostream & ComposPipeline::Ostream(ostream & os) const
{
    // append ComposPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
40  os << endl << "  chipIndex           = " << chipIndex;
    os << endl << "  results.coxel       = " << results.coxel;

    // return complete os
    return os;
45  } // Ostream

50  /////////////////////////////////
// show/set data & data properties

```

45

```

//  

// - local show/set functions  

5  

void ComposPipeline::GlobalSetup(const int setNumOfChips,  

                                  const int setNumOfPipelinesPerChip)  

{  

    numOfChips          = setNumOfChips;  

    numOfPipelinesPerChip = setNumOfPipelinesPerChip;  

} // GlobalSetup  

void ComposPipeline::LocalSetup(const int setChipIndex,  

                                const int setPipelineIndex,  

                                ComposStage & composStage)  

{  

    chipIndex = setChipIndex;  

    pipelineIndex = setPipelineIndex;  

    inputs.shadel = &(composStage.inputs.shadel[pipelineIndex]);  

    inputs.coxel = &(composStage.inputs.coxel[pipelineIndex]);  

    inputs.weightsXYZ = &(composStage.inputs.weightsXYZ[pipelineIndex]);  

    inputs.perChipControlFlags  

        = composStage.inputs.perChipControlFlags;  

    inputs.perPipelineControlFlags  

        = &(composStage.inputs.perPipelineControlFlags[pipelineIndex]);  

    results.coxel = &(composStage.tempResults.coxel[pipelineIndex]);  

    results.perPipelineControlFlags  

        = &(composStage.tempResults.perPipelineControlFlags[pipelineIndex]);  

    results.perChipControlFlags  

        = &(composStage.tempResults.perChipControlFlags);  

    cube4 = composStage(cube4;  

} // LocalSetup  

void ComposPipeline::PerFrameSetup()  

{  

    // reset pipeline registers already done in ComposStage  

} // PerFrameSetup  

///////////////////////////////  

// local computation functions  

45  

void ComposPipeline::RunForOneClockCycle()  

{
    if (cube4->compositingStyle == BackToFront) {

```

50

55

```

        46
    results.coxel->r = (1.0-      inputs.shadel->a)*inputs.coxel->r +
inputs.shadel->r;
    results.coxel->g = (1.0-inputs.shadel->a)*inputs.coxel->g +
inputs.shadel->g;
    results.coxel->b = (1.0-inputs.shadel->a)*inputs.coxel->b +
inputs.shadel->b;
    results.coxel->a = (1.0-inputs.shadel->a)*inputs.coxel->a +
inputs.shadel->a;
10   }

    if (cube4->compositingStyle == FrontToBack) {
        results.coxel->r = (1.0-inputs.coxel->a)*inputs.shadel->r +
inputs.coxel->r;
    results.coxel->g = (1.0-inputs.coxel->a)*inputs.shadel->g +
inputs.coxel->g;
    results.coxel->b = (1.0-inputs.coxel->a)*inputs.shadel->b +
inputs.coxel->b;
    results.coxel->a = (1.0-inputs.coxel->a)*inputs.shadel->a +
inputs.coxel->a;
15   }

// if debug: hand inputs through without change
results.coxel->r = inputs.shadel->r;
results.coxel->g = inputs.shadel->g;
20   results.coxel->b = inputs.shadel->b;
    results.coxel->a = inputs.shadel->a;
} // RunForOneClockCycle

30   /////////////////////////////////
// internal utility functions

// end of ComposPipeline.C
:::::::::::
35 cube4/ComposPipeline.h
:::::::::::
// ComposPipeline.h
// (c) Ingmar Bitter '97

40 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// Premultiplied Colors (C implies A*C)
//
// Back-to-Front
45 // Cacc = (1-Anew)Cacc + Cnew
// Aacc = (1-Anew)Aacc + Anew
//
// Front-to-Back
// Cacc = (1-Aacc)Cnew + Cacc
// Aacc = (1-Aacc)Anew + Aacc
50

```

```

      47
#ifndef _ComposPipeline_h_    //      prevent multiple includes
#define _ComposPipeline_h_
5
#include "Misc.h"
#include "Object.h"
#include "Shadel.h"
#include "Coxel.h"
#include "Control.h"

10
class ComposStage;
class Cube4;

15
class ComposPipelineInputs {
public: // pointers
    Shadel *shadel;
    Coxel *coxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
20
    PerPipelineControlFlags *perPipelineControlFlags;
};

25
class ComposPipelineResults {
public: // pointers
    Coxel *coxel;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

30
class ComposPipeline : virtual public Object {
public:

35
    static void Demo ();
    // constructors & destructors
    ComposPipeline ();
    ~ComposPipeline ();

40
    // show/set data & data properties
    // - class Object requirements
    virtual ostream & Ostream (ostream & ) const;

45
    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
    const int setNumOfPipelinesPerChip);

    virtual void LocalSetup(const int setChipIndex,
    const int setPipelineIndex,
    ComposStage & composStage);

50
55

```

```

48
    virtual void PerFrameSetup();
    // local computation functions
    virtual void RunForOneClockCycle();

5   public:
        ComposPipelineInputs inputs;
        ComposPipelineResults results;

10  protected:
        static int      numOfChips, numOfPipelinesPerChip;
        static Cube4 *cube4;
        int           chipIndex, pipelineIndex;    // only for debugging purpose
};

15 #include "ComposStage.h"
#include "Cube4.h"

#ifndef _ComposPipeline_h_
:::::::::::
20 cube4/ComposSelXPipeline.C
:::::::::::
// ComposSelXPipeline.C
// (c) Ingmar Bitter '97

25 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposSelXPipeline.h"

void ComposSelXPipeline::Demo()
{
30     ComposSelXPipeline composSelX;
    cout << endl << "Demo of class " << typeid(composSelX).name();
    cout << endl << "size : " << sizeof(ComposSelXPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposSelXPipeline composSelX; = " << composSelX;
    cout << endl << "End of demo of class " << typeid(composSelX).name() << endl;
} // Demo

40 //////////////////////////////////////////////////////////////////
// constructors & destructors
// static first init
int ComposSelXPipeline::numOfChips          = 0;
int ComposSelXPipeline::numOfPipelinesPerChip = 0;
int ComposSelXPipeline::blockSize            = 0;
Cube4 *ComposSelXPipeline::cube4             = 0;

45 ComposSelXPipeline::ComposSelXPipeline()
{
    // Number of clock cycles to process a partial beam inside a block
    int partialBeamDelay = blockSize / numOfPipelinesPerChip;
}

```

55

49

```

5   // Delay for communication: 16 bits, 4 bits / cycle, double frequency => 2
cycles
6   int communicationDelay = 2;

7   partialBeamCoxelFiFo.SetSize(partialBeamDelay);
8   partialBeamWeightsFiFo.SetSize(partialBeamDelay);
9   partialBeamPerPipelineControlFlagsFiFo.SetSize(partialBeamDelay);
10  partialBeamPerChipControlFlagsFiFo.SetSize(partialBeamDelay);

11  communicationDelayCoxelFiFo.SetSize(communicationDelay);
12  communicationDelayWeightsFiFo.SetSize(communicationDelay);
13  communicationDelayPerPipelineControlFlagsFiFo.SetSize(communicationDelay);
14  communicationDelayPerChipControlFlagsFiFo.SetSize(communicationDelay);
15 } // constructor

16
17 ComposSelXPipeline::~ComposSelXPipeline()
18 {
19 } // destructor

20
21
22 //////////////////////////////////////////////////////////////////
23 // show/set data & data properties
24
25
26 ostream & ComposSelXPipeline::Ostream(ostream & os) const
27 {
28   // append ComposSelXPipeline info to os
29   os << typeid(*this).name() << "@" << (void *) this;
30   os << endl << " numOfChips           = " << numOfChips;
31   os << endl << " numOfPipelinesPerChip = " << numofPipelinesPerChip;
32   os << endl << " chipIndex              = " << chipIndex;

33   // return complete os
34   return os;
35 }
36 } // Ostream

37
38 //////////////////////////////////////////////////////////////////
39 // show/set data & data properties
40 //
41 // - local show/set functions

42
43 void ComposSelXPipeline::GlobalSetup(const int setNumOfChips,
44                                     const int setNumOfPipelinesPerChip,
45                                     const int setBlockSize)
46 {
47   numOfChips          = setNumOfChips;
48   numofPipelinesPerChip = setNumOfPipelinesPerChip;
49   blockSize           = setBlockSize;
50 }
```

55

```

      50
} // GlobalSetup

5   void ComposSelXPipeline::LocalSetup(const int setChipIndex,
                                         const int setPipelineIndex,
                                         ComposSelXStage & composSelXStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

10  inputs.coxelMiddle = &(composSelXStage.computation.coxel[pipelineIndex]);
    inputs.coxelRight = &(composSelXStage.computation.coxel[pipelineIndex+1]);
    inputs.weightsXYZ = &(composSelXStage.computation.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = &(composSelXStage.computation.perChipControlFlags);
    inputs.perPipelineControlFlags
        = &(composSelXStage.computation.perPipelineControlFlags[pipelineIndex]);

15  results.coxel = &(composSelXStage.results.coxel[pipelineIndex]);
    results.weightsXYZ = &(composSelXStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(composSelXStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(composSelXStage.results.perChipControlFlags);

20  cube4 = composSelXStage(cube4);
} // LocalSetup

void ComposSelXPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in TriLinXStage
    partialBeamCoxelFiFo.Preset(*(&inputs.coxelMiddle));
    partialBeamWeightsFiFo.Preset(*(&inputs.weightsXYZ));

    partialBeamPerPipelineControlFlagsFiFo.Preset(*(&inputs.perPipelineControlFlags));
    ;
    partialBeamPerChipControlFlagsFiFo.Preset(*(&inputs.perChipControlFlags));

    communicationDelayCoxelFiFo.Preset(*(&inputs.coxelMiddle));
    communicationDelayWeightsFiFo.Preset(*(&inputs.weightsXYZ));

    communicationDelayPerPipelineControlFlagsFiFo.Preset(*(&inputs.perPipelineControl
Flags));
    ;

    communicationDelayPerChipControlFlagsFiFo.Preset(*(&inputs.perChipControlFlags));
} // PerFrameSetup

45  /////////////////////////////////
// local computation functions

void ComposSelXPipeline::RunForOneClockCycle()

```

50

55

```

(
    if (cube4->cubeMode == Cube4Classic) {
5
        if (inputs.perChipControlFlags->xStep == Right) {
            // data comes from the left (which happens in ComposLinX)
            // L      M      R (data src)
            // R      M      L (step)
10       // o-----o
            *(results.coxel) = *(inputs.coxelMiddle);
        }
        else if (inputs.perChipControlFlags->xStep == Left) {
            // data comes form the right
            // L      M      R (data src)
            // R      M      L (step)
            // o      o<----o
15       *(results.coxel) = *(inputs.coxelRight);
        }
        else if (inputs.perChipControlFlags->xStep == Middle) {
20       // data goes straight
            // L      M      R (data src)
            // R      M      L (step)
            // o      o      o
            *(results.coxel) = *(inputs.coxelMiddle);
        }
25       else { ERROR( "Wrong xstep in ComposSelXPipeline!" ); }
    }

    else if (cube4->cubeMode == Cube4Light) {
30       // selecting either the left or right coxel,
       // depending on sign of x weight

        if (inputs.weightsXYZ->X() <= 0) {
35       // data comes from the left or goes straight
            // L      M      R (weight / data src)
            // o-----o
            *(results.coxel) = *(inputs.coxelMiddle);
        }
        else { // inputs.weightsXYZ->X() > 0
40       // data comes from the right
            // L      M      R (weight / data src)
            // o      o<----o
            *(results.coxel) = *(inputs.coxelRight);
        }
45       }
    } // RunForOneClockCycle

///////////////////////////////
// internal utility functions
50

// end of ComposSelXPipeline.C

```

52

```

::::::::::
cube4/ComposSelXPipeline.h
::::::::::
// ComposSelXPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// 10 #ifndef _ComposSelXPipeline_h_           // prevent multiple includes
#define _ComposSelXPipeline_h_

// 15 #include "Misc.h"
#include "Object.h"
#include "ModInt.h"
#include "Coxel.h"
#include "FiFo.h"
#include "Control.h"
// 20 #include "FixPointNumber.h"
#include "Vector3D.h"

typedef Vector3D<FixPointNumber> FixPointVector3D;

// 25 class ComposSelXStage;
class Cube4;

class ComposSelXPipelineInputs {
public: // pointers
    // 30 Coxel *coixelMiddle;
    Coxel *coixelRight;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

// 35

class ComposSelXPipelineResults {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

// 40

class ComposSelXPipeline : virtual public Object {
public:

    static void      Demo ();

// 45    // constructors & destructors
    ComposSelXPipeline ();
    ~ComposSelXPipeline ();
};

// 50

```

55

```

// show/set data & data properties
5   // - class Object requirements
    virtual ostream & Ostream (ostream & ) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
                        const int setNumOfPipelinesPerChip,
10                      const int setBlockSize);

virtual void LocalSetup(const int setChipIndex,
                       const int setPipelineIndex,
                       ComposSelXStage & composSelXStage);
15   virtual void PerFrameSetup();
// local computation functions
virtual void RunForOneClockCycle();

public:
  ComposSelXPipelineInputs inputs;
20  ComposSelXPipelineResults results;

protected:
  Fifo<Coxel>           partialBeamCoxelFifo;
  Fifo<FixPointVector3D> partialBeamWeightsFifo;
  Fifo<PerPipelineControlFlags> partialBeamPerPipelineControlFlagsFifo;
25  Fifo<PerChipControlFlags> partialBeamPerChipControlFlagsFifo;

  Fifo<Coxel>           communicationDelayCoxelFifo;
  Fifo<FixPointVector3D> communicationDelayWeightsFifo;
  Fifo<PerPipelineControlFlags> communicationDelayPerPipelineControlFlagsFifo;
30  Fifo<PerChipControlFlags> communicationDelayPerChipControlFlagsFifo;

  static int   numOfChips, numOfPipelinesPerChip, blockSize;
  static Cube4 *cube4;
  int         chipIndex, pipelineIndex;

35   friend class ComposSelXStage;
  friend class Cube4;
};

#include "ComposSelXStage.h"
40 #include "Cube4.h"

#endif // _ComposSelXPipeline_h_
:::::::::::
cube4/ComposSelXStage.C
:::::::::::
45 // ComposSelXStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

```

50

55

54

```

#include "ComposSelXStage.h"

5   void ComposSelXStage::Demo()
{
    ComposSelXStage composSelX;
    cout << endl << "Demo of class " << typeid(composSelX).name();
    cout << endl << "size : " << sizeof(ComposSelXStage) << " Bytes";
    cout << endl << "public member functions:";
10  cout << endl << "ComposSelXStage composSelX; = " << composSelX;
    cout << endl << "End of demo of class " << typeid(composSelX).name() << endl;
} // Demo

15 //////////////////////////////////////////////////////////////////
// constructors & destructors

// static first init
int ComposSelXStage::numOfChips           = 0;
int ComposSelXStage::numOfPipelinesPerChip = 0;
20 Cube4 *ComposSelXStage::cube4 = 0;

ComposSelXStage::ComposSelXStage()
{
    composSelXPipeline = new ComposSelXPipeline [numOfPipelinesPerChip];

25   computation.coxel
        = new Coxel [numOfPipelinesPerChip+1];
    computation.weightsXYZ
        = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    computation.perPipelineControlFlags
        = new PerPipelineControlFlags [numOfPipelinesPerChip+1]; // +1 just
30   for debugging

    results.coxel
        = new Coxel [numOfPipelinesPerChip];
    results.weightsXYZ
        = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags
        = new PerPipelineControlFlags [numOfPipelinesPerChip];
} // defaultconstructor

40 ComposSelXStage::~ComposSelXStage()
{
    if (composSelXPipeline) { delete composSelXPipeline; composSelXPipeline=0; }

    if (computation.coxel) {
        delete computation.coxel;
45     computation.coxel=0;
    }
    if (computation.weightsXYZ) {
        delete computation.weightsXYZ;
        computation.weightsXYZ=0;
    }
50
}

```

55

55

```

    }
    if (computation.perPipelineControlFlags) {
        delete computation.perPipelineControlFlags;
        computation.perPipelineControlFlags=0;
    }

    if (results.coxel) { delete results.coxel; results.coxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0; }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

15
///////////////////////////////
// show/set data & data properties

20 ostream & ComposSelXStage::Ostream(ostream & os) const
{
    // append ComposSelXStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips      = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
25    os << endl << "  chipIndex       = " << chipIndex;

    // return complete os
    return os;

30 } // Ostream

///////////////////////////////
// show/set data & data properties
//
35 // - local show/set functions

void ComposSelXStage::GlobalSetup(const int setNumOfChips,
                                   const int setNumOfPipelinesPerChip,
                                   Cube4 *setCube4)
40 {
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    cube4               = setCube4;
    ComposSelXPipeline::GlobalSetup(numOfChips,
45
                                   numOfPipelinesPerChip,
                                   cube4->blockSize);
} // GlobalSetup
50

```

55

```

void ComposSelXStage::LocalSetup(const int setChipIndex)
5
{
    chipIndex(setChipIndex, numOfChips);
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composSelXPipeline[p].LocalSetup(chipIndex, p, *this);
    }
} // LocalSetup

10

void ComposSelXStage::PerFrameSetup()
{
    // reset pipeline registers
15
    int p;
    readBufferCoxel = 0;
    communicationCoxel = 0;
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        computation.coxel[p] = 0;
        computation.weightsXYZ[p](0,0,0);
20
        computation.perPipelineControlFlags[p].Reset();
    }
    computation.coxel[numOfPipelinesPerChip] = 0;

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.coxel[p] = cube4->backgroundCoxel;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
25
    }
    results.perChipControlFlags.Reset();

30
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        composSelXPipeline[p].PerFrameSetup();
    }
} // PerFrameSetup

35
///////////////////////////////
// local computation functions

void ComposSelXStage::CommunicateForOneClockCycle()
40
{
    ///////////////////////////////
    // fill communication register

    // ****
    // scrap following for loop when ComposBuffer is ok !!!!!!!!
45
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        inputs.coxel[p].r = inputs.perPipelineControlFlags[p].voxelPosXYZ.X();
        inputs.coxel[p].g = inputs.perPipelineControlFlags[p].voxelPosXYZ.Y();
        inputs.coxel[p].b = inputs.perPipelineControlFlags[p].voxelPosXYZ.Z();
        inputs.coxel[p].a = inputs.perPipelineControlFlags[p].voxelPosXYZ.Z();
50
    }
}

```

```

5 // only sent data at beginning of a block
if (inputs.perChipControlFlags->xBlockStart) {

    // write readbuffer to communication register
    cube4->composSelX[chipIndex-1].communicationCoxel =
        cube4->composSelX[chipIndex-1].readBufferCoxel;
    cube4->composSelX[chipIndex-1].communicationPerPipelineControlFlags =
        cube4->composSelX[chipIndex-1].readBufferPerPipelineControlFlags;

10   // leftmost chip sends data as early as possible
    if (inputs.perChipControlFlags->leftmostChip) {
        cube4->composSelX[chipIndex-1].readBufferCoxel =
            inputs.coxel[0];
        cube4->composSelX[chipIndex-1].readBufferPerPipelineControlFlags =
            inputs.perPipelineControlFlags[0];
    }
    // remaining chips send (blockSize/numOfPipelinesPerChip) later
20   else {
        cube4->composSelX[chipIndex-1].readBufferCoxel =
            composSelXPipeline[0].partialBeamCoxelFiFo.Peek(0);
        cube4->composSelX[chipIndex-1].readBufferPerPipelineControlFlags =
            composSelXPipeline[0].partialBeamPerPipelineControlFlagsFiFo.Peek(0);
    }
25 }

} // CommunicateForOneClockCycle

30 void ComposSelXStage::RunForOneClockCycle()
{
    /*
35     +---+-----+
     | 8 | 8 | 8 | 8 |
     +---+-----+           +---+-----+           inputs
                                |
                                +---+-----+
                                | 5 | <--| ? | <-+
                                +---+-----+
                                | 3 |           ^
                                +---+-----+
40     +---+           |
                                +---+-----+           read
                                | 5 |           buffer
                                +---+-----+
                                | 3 |           communication
                                +---+-----+           register
                                |
                                +---+-----+
                                | 7 | 7 | 7 | 7 |
                                +---+-----+
                                | 6 | 6 | 6 | 6 |
                                +---+-----+
45     +---+-----+
                                +---+-----+           partial
                                | 7 | 7 | 7 | 7 |           beam
                                +---+-----+           FiFo
                                +---+-----+
                                | 6 | 6 | 6 | 6 |
                                +---+-----+
                                +---+-----+
                                +---+-----+
                                | 5 | 5 | 5 | 5 |
                                +---+-----+
                                | 4 | 4 | 4 | 4 |
                                +---+-----+ V
                                +---+-----+           communication
                                | 5 | 5 | 5 | 5 |           delay
                                +---+-----+           FiFo
                                +---+-----+
50     +---+-----+

```

```

      |           +--+
      |           |   58
      |           +--+
      |           |   +----->| ? |
      |           |           +--+
      |           |           |   |
      |           |           V
      |           +-----+ +-----+
      |           | 3 | 3 | 3 | 3 | 4 |
      |           +-----+ +-----+
10      +-----+ +-----+
      |           | 2 | 2 | 2 | 2 |
      |           +-----+ +-----+
      +-----+
*/          +-----+
           | 2 | 2 | 2 | 2 |
           +-----+
           computation
           register
           results
15      int p;

/////////////////////////////
// computation
results.perChipControlFlags = computation.perChipControlFlags;
20      for (p=0; p<numOfPipelinesPerChip; ++p) {
    results.weightsXYZ[p] = computation.weightsXYZ[p];
    results.perPipelineControlFlags[p] = computation.perPipelineControlFlags[p];
    composSelXPipeline[p].RunForOneClockCycle();
}

25      //////////////////
// fill computation registers

// for rightmost computation register choose between
// communicationDelayCoxelFiFo and communication register
30      if (composSelXPipeline[0].RunForOneClockCycle() {
        communicationDelayPerChipControlFlagsFiFo.Peek(1).xBlockStart) {
    // at end of block copy from right chip
    computation.coxel[numOfPipelinesPerChip]
        = communicationCoxel;
35      computation.perPipelineControlFlags[numOfPipelinesPerChip]
        = communicationPerPipelineControlFlags;
}
else {
    // within block copy from leftmost pipeline
    computation.coxel[numOfPipelinesPerChip]
        = composSelXPipeline[0].communicationDelayCoxelFiFo.Peek(1);
    computation.perPipelineControlFlags[numOfPipelinesPerChip]
        =
composSelXPipeline[0].communicationDelayPerPipelineControlFlagsFiFo.Peek(1);
}

40      // always copy communicationDelayFiFo to remaining computation registers
for (p=0; p<numOfPipelinesPerChip; ++p) {
    computation.coxel[p]
        = composSelXPipeline[p].communicationDelayCoxelFiFo.Read();
    computation.weightsXYZ[p]
50

```

59

```

      =
composSelXPipeline[p].communicationDelayWeightsFiFo.Read();
    computation.perPipelineControlFlags[p]
    =
composSelXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Read();
}
computation.perChipControlFlags
  = composSelXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Read();

10 /////////////////
// Write to communication delay Fifo
for (p=0; p<numOfPipelinesPerChip; ++p) {
    composSelXPipeline[p].communicationDelayCoxelFiFo.Write
      (composSelXPipeline[p].partialBeamCoxelFiFo.Read());
    composSelXPipeline[p].communicationDelayWeightsFiFo.Write
      (composSelXPipeline[p].partialBeamWeightsFiFo.Read());
    composSelXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Write
      (composSelXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Read());
}

15 composSelXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Write
  (composSelXPipeline[0].partialBeamPerChipControlFlagsFiFo.Read());

20 /////////////////
// Write to partial beam Fifo
for (p=0; p<numOfPipelinesPerChip; ++p) {
    composSelXPipeline[p].partialBeamCoxelFiFo.Write
      (inputs.coxel[p]);
    composSelXPipeline[p].partialBeamWeightsFiFo.Write
      (inputs.weightsXYZ[p]);
    composSelXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Write
      (inputs.perPipelineControlFlags[p]);
}

25 composSelXPipeline[0].partialBeamPerChipControlFlagsFiFo.Write
  (*inputs.perChipControlFlags);

30 } // RunForOneClockCycle

35 // end of ComposSelXStage.C
:::::::::::
cube4/ComposSelXStage.h
:::::::::::
40 // ComposSelXStage.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45 #ifndef _ComposSelXStage_h_    // prevent multiple includes
#define _ComposSelXStage_h_

50 #include "Misc.h"
#include "Object.h"
#include "Coxel.h"

```

55

60

```

#include "Control.h"
#include "ComposSelXPipeline.h"
#include "FixPointNumber.h"

5      class Cube4;

class ComposSelXStageInputs {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
15     };

class ComposSelXStageResults {
public: // arrays
20     Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

25     class ComposSelXStage : virtual public Object {
public:

    static void Demo ();

30     // constructors & destructors
    ComposSelXStage ();
    ~ComposSelXStage ();

    // show/set data & data properties
    // - class Object requirements
35     virtual ostream & Ostream (ostream &) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           Coxel *setCube4);
    virtual void LocalSetup (const int setChipIndex);
    virtual void PerFrameSetup ();
45     // local computation functions
    virtual void CommunicateForOneClockCycle();
    virtual void RunForOneClockCycle();

    public:
50     ComposSelXPipeline *composSelXPipeline;
     ComposSelXStageInputs inputs;
     ComposSelXStageResults computation;
     ComposSelXStageResults results;
}

```

55

```

61
protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;

    ModInt      chipIndex;
    Coxel       readBufferCoxel;
    PerPipelineControlFlags readBufferPerPipelineControlFlags; // only for
debugging
    Coxel       communicationCoxel;
    PerPipelineControlFlags communicationPerPipelineControlFlags; // only for
debugging

    friend class ComposSelXPipeline;
    friend class Cube4;
};

#include "Cube4.h"

#endif // _ComposSelXStage_h_
:::::::::::
cube4/ComposSelYPipeline.C
:::::::::::
// ComposSelYPipeline.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposSelYPipeline.h"

30 void ComposSelYPipeline::Demo()
{
    ComposSelYPipeline composSelY;
    cout << endl << "Demo of class " << typeid(composSelY).name();
    cout << endl << "size : " << sizeof(ComposSelYPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ComposSelYPipeline composSelY; = " << composSelY;
    cout << endl << "End of demo of class " << typeid(composSelY).name() << endl;
} // Demo

40 /////////////////////////////////
// constructors & destructors

// static first init
int ComposSelYPipeline::numOfChips          = 0;
int ComposSelYPipeline::numOfPipelinesPerChip = 0;
45 int ComposSelYPipeline::blockSize          = 0;
Cube4 *ComposSelYPipeline::cube4            = 0;

ComposSelYPipeline::ComposSelYPipeline()
{
} // constructor
50

```

```

5   ComposSelYPipeline::~ComposSelYPipeline()
{
} // destructor

10  //////////////////////////////////////////////////////////////////
// show/set data & data properties

15  ostream & ComposSelYPipeline::Ostream(ostream & os) const
{
    // append ComposSelYPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;
20
    // return complete os
    return os;
}

25  } // Ostream

25  //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

30

35  void ComposSelYPipeline::GlobalSetup(const int setNumOfChips,
                                         const int setNumOfPipelinesPerChip,
                                         const int setBlockSize)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
} // GlobalSetup

40

40  void ComposSelYPipeline::LocalSetup(const int setChipIndex,
                                         const int setPipelineIndex,
                                         ComposSelYStage & composSelYStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.coxel = &(composSelYStage.inputs.coxel[pipelineIndex]);
    inputs.weightsXYZ = &(composSelYStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
50
      = composSelYStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
}

```

63

```

    =
&(composSelYStage.inputs.perPipelineControlFlags[pipelineIndex]);
5   results.coxel = &(composSelYStage.results.coxel[pipelineIndex]);
results.weightsXYZ = &(composSelYStage.results.weightsXYZ[pipelineIndex]);
results.perPipelineControlFlags
= &(composSelYStage.results.perPipelineControlFlags[pipelineIndex]);
results.perChipControlFlags
= &(composSelYStage.results.perChipControlFlags);

10  cube4 = composSelYStage(cube4;
} // LocalSetup

15 void ComposSelYPipeline::PerFrameSetup()
{
    // resize fifo's according to dataset size
    // step delay for a y-step within a block
    beamDelay = (cube4->datasetSizeXYZ.X())
20
    (numOfChips*numOfPipelinesPerChip);

    // step delay for a y-step between blocks
    blockBeamDelay = beamDelay * blockSize;

25    beamCoxelFiFo.SetSize(beamDelay);
    beamWeightsFiFo.SetSize(beamDelay);
    beamPerPipelineControlFlagsFiFo.SetSize(beamDelay);
    beamPerChipControlFlagsFiFo.SetSize(beamDelay);

    blockBeamCoxelFiFo.SetSize(blockBeamDelay);
30    blockBeamWeightsFiFo.SetSize(blockBeamDelay);
    blockBeamPerPipelineControlFlagsFiFo.SetSize(blockBeamDelay);
    blockBeamPerChipControlFlagsFiFo.SetSize(blockBeamDelay);

    beamCoxelFiFo.Preset(*(inputs.coxel));
35    beamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    beamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    blockBeamCoxelFiFo.Preset(*(inputs.coxel));
40    blockBeamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    blockBeamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
    blockBeamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    readBigFiFoCounter = readSmallFiFoCounter =
45    writeBigFiFoCounter = writeSmallFiFoCounter = -1;
} // PerFrameSetup

50
////////////////////////////////////////////////////////////////
// local computation functions

```

55

```

void ComposSelVPipeline::RunForOneClockCycle()
{
    // reset counters
    if (inputs.perChipControlFlags->volumeStart ||
        ((readBigFiFoCounter == 0) &&
         (readSmallFiFoCounter == 0) &&
         (writeBigFiFoCounter == 0) &&
         (writeSmallFiFoCounter == 0))) {
        10
        readBigFiFoCounter = beamDelay;
        readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;

        writeBigFiFoCounter = readBigFiFoCounter;
        writeSmallFiFoCounter = readSmallFiFoCounter;
    }

    /////////////////////////////////
    // first read from Fifos into results register

    20
    // at start of block read from big Fifo
    if (readBigFiFoCounter > 0) {
        blockBeamCoxelFiFo.Read(fifoCoxel);
        blockBeamWeightsFiFo.Read(*(results.weightsXYZ));
    }

    25
    blockBeamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlag
s));

    blockBeamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
    --readBigFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0)
    30
    cout<<"R"<<*(results.coxel)<<volumeSliceGradientFiFo<<endl;
}

    // in middle and at end of block read from small Fifo
    else if (readSmallFiFoCounter > 0) {
        beamCoxelFiFo.Read(fifoCoxel);
        beamWeightsFiFo.Read(*(results.weightsXYZ));
    }

    35
    beamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
    --readSmallFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0)      cout<<"r";
}

    ///////////////////////////////
    // buffer writing

    45
    ///////////////////////////////
    // now write to Fifos from inputs register

    // at start and in middle of block write to small Fifo
    if (writeSmallFiFoCounter > 0) {
}

```

50

55

```

beamCoxelFiFo.Write(*(inputs.coxel));
beamWeightsFiFo.Write(*(inputs.weightsXYZ) );
5

beamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
beamPerChipControlFlagsFiFo.Write(
*(inputs.perChipControlFlags));
--writeSmallFiFoCounter;
10
// if (chipIndex == 0 && pipelineIndex == 0) cout<<"w";
}

// at end of block write to big FiFo of next chip
else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
    ComposSelYPipeline *next(&cube4-
15
>composSelY[c].composSelYPipeline[p]);
    next->blockBeamCoxelFiFo.Write(*(inputs.coxel));
    next->blockBeamWeightsFiFo.Write(*(inputs.weightsXYZ) );
    next-
20
>blockBeamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
    next-
>blockBeamPerChipControlFlagsFiFo.Write(*(inputs.perChipControlFlags));
    --writeBigFiFoCounter;
    //if (chipIndex == 0 && pipelineIndex == 0) cout<<"W";
25
}

///////////////////////////////
// select results from Fifo or inputs depending on yStep or y-weight

30
if (cube4->cubeMode == Cube4Classic) {

    if (inputs.perChipControlFlags->yStep == Down) {
        // data comes from the top
        // T      M      B  (data src)
        // B      M      T  (step)
        // o---->o      o
35
        *(results.coxel) = fifoCoxel;
    }
    else if (inputs.perChipControlFlags->yStep == Up) {
        // data comes from the bottom
        // T      M      B  (data src)
        // B      M      T  (step)
        // o      o<----o
        *(results.coxel) = *(inputs.coxel);
    }
40
    if (inputs.perChipControlFlags->yStep == Middle) {
        // data goes straight
        // T      M      B  (data src)
        // B      M      T  (step)
        // o      o      o
        *(results.coxel) = fifoCoxel;
    }
45
}
50

```

```

66
}
else { ERROR( "Wrong ystep in ComposSelYPipeline!" ); }
5
    }

else if (cube4->cubeMode == Cube4Light) {

    if (inputs.weightsXYZ->Y() <= 0) {
10   // data comes from the top or goes straight
    // T      M      B  (weight / data src)
    // o---->o      o
    *(results.coxel) = fifoCoxel;
    }

    else { // inputs.weightsXYZ->Y() > 0
15   // data comes from the bottom
    // T      M      B  (weight / data src)
    // o      o<----o
    *(results.coxel) = *(inputs.coxel);
    }
20
}

} // RunForOneClockCycle

///////////////////////////////
25 // internal utility functions

// end of ComposSelYPipeline.C
:::::::::::
cube4/ComposSelYPipeline.h
30 :::::::::::
// ComposSelYPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
35 // America, Inc., 1997, All rights reserved.

#ifndef _ComposSelYPipeline_h_      // prevent multiple includes
#define _ComposSelYPipeline_h_

40 #include "Misc.h"
#include "Object.h"
#include "FiFo.h"
#include "Coxel.h"
#include "Control.h"
#include "FixPointNumber.h"
45

typedef Vector3D<FixPointNumber> FixPointVector3D;

class ComposSelYStage;
class Cube4;
50 class ComposSelYPipelineInputs {
public: // pointers

```

55

```

5           Coxel *coixel;
5           Vector3D<FixPointNumber> *weightsXYZ;
5           PerChipControlFlags      *perChipControlFlags;
5           PerPipelineControlFlags *perPipelineControlFlags;
5       };

10      class ComposSelYPipelineResults {
10      public: // pointers
10          Coxel *coixel;
10          Vector3D<FixPointNumber> *weightsXYZ;
10          PerChipControlFlags      *perChipControlFlags;
10          PerPipelineControlFlags *perPipelineControlFlags;
10      };

20      class ComposSelYPipeline : virtual public Object {
20      public:
20          static void      Demo ();
20
20          // constructors & destructors
20          ComposSelYPipeline ();
20          ~ComposSelYPipeline ();
20
20          // show/set data & data properties
20          // - class Object requirements
20          virtual ostream & Ostream (ostream & ) const;
20
20          // - local show/set functions
20          static void GlobalSetup (const int setNumOfChips,
20                                  const int setNumOfPipelinesPerChip,
20                                  const int setBlockSize);
20
20          virtual void LocalSetup(const int setChipIndex,
20                                 const int setPipelineIndex,
20                                 ComposSelYStage & composSelYStage);
20          virtual void PerFrameSetup();
20          // local computation functions
20          virtual void RunForOneClockCycle();
20
20      public:
20          ComposSelYPipelineInputs  inputs;
20          ComposSelYPipelineResults results;
20
20      protected:
20          Fifo<Coxel>              beamCoxelFifo;
20          Fifo<FixPointVector3D>    beamWeightsFifo;
20          Fifo<PerPipelineControlFlags> beamPerPipelineControlFlagsFifo;
20          Fifo<PerChipControlFlags>  beamPerChipControlFlagsFifo;
20
20          Fifo<Coxel>              blockBeamCoxelFifo;
20          Fifo<FixPointVector3D>    blockBeamWeightsFifo;
20
55

```

```

68
FIFO<PerPipelineControlFlags>          blockBeamPerPipelineControlFlagsFIFO;
FIFO<PerChipControlFlags>           blockBeamPerChipControlFlagsFIFO;

5   Coxel fifoCoxel;

    int beamDelay, blockBeamDelay;

    int  readSmallFiFoCounter, readBigFiFoCounter;
10   int  writeSmallFiFoCounter, writeBigFiFoCounter;

    static int  numOfChips, numOfPipelinesPerChip, blockSize;
    static Cube4 *cube4;
    int         chipIndex, pipelineIndex;

15   friend class Cube4;
}; // ComposSelYPipeline

#include "ComposSelYStage.h"
#include "Cube4.h"

20 #endif      // _ComposSelYPipeline_h_
:::::::::::
cube4/ComposSelYStage.C
:::::::::::
// ComposSelYStage.C
// (c) Ingmar Bitter '97

25 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ComposSelYStage.h"
30 void ComposSelYStage::Demo()
{
    ComposSelYStage composSelY;
    cout << endl << "Demo of class " << typeid(composSelY).name();
    cout << endl << "size : " << sizeof(ComposSelYStage) << " Bytes";
35   cout << endl << "public member functions:";
    cout << endl << "ComposSelYStage composSelY; = " << composSelY;
    cout << endl << "End of demo of class "<< typeid(composSelY).name() <<
endl;
} // Demo

40 /////////////////
// constructors & destructors

45 // static first init
int ComposSelYStage::numOfChips        = 0;
int ComposSelYStage::numOfPipelinesPerChip = 0;
Cube4 *ComposSelYStage::cube4 = 0;

ComposSelYStage::ComposSelYStage()

50

```

```

5
{
    composSelYPipeline = new ComposSelYPipeline [numOfPipelinesPerChip];
    results.coxel = new Coxel [numOfPipelinesPerChip + 1];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
    [numOfPipelinesPerChip];
} // defaultconstructor

10
ComposSelYStage::~ComposSelYStage()
{
    if (composSelYPipeline) { delete composSelYPipeline; composSelYPipeline=0;
}
    if (results.coxel)      { delete results.coxel;      results.coxel=0;
}
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
}
} // destructor

25
////////////////////////////// show/set data & data properties

30
ostream & ComposSelYStage::Ostream(ostream & os) const
{
    // append ComposSelYStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex           = " << chipIndex;

    // return complete os
35
    return os;
}

40
////////////////////////////// show/set data & data properties
// - local show/set functions

45
void ComposSelYStage::GlobalSetup(const int setNumOfChips,
{
    const int setNumOfPipelinesPerChip,
    const int setBlockSize,
}
50

```

```

        Cube4 *setCube4)
5      {
        numOfChips          = setNumOfChips;
        numOfPipelinesPerChip = setNumOfPipelinesPerChip;
        ComposSelYPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
                                         setBlockSize);
10     cube4 = setCube4;
    } // GlobalSetup

    void ComposSelYStage::LocalSetup(const int setChipIndex)
15   {
        chipIndex = setChipIndex;
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            composSelYPipeline[p].LocalSetup(chipIndex,p,*this);
        }
    } // LocalSetup
20

    void ComposSelYStage::PerFrameSetup()
    {
        int p;
        // reset pipeline registers
25      for (p=0; p<numOfPipelinesPerChip; ++p) {
            results.coxel[p] = cube4->backgroundCoxel;
            results.weightsXYZ[p](0,0,0);
            results.perPipelineControlFlags[p].Reset();
        }
        results.perChipControlFlags.Reset();

        for (p=0; p<numOfPipelinesPerChip; ++p) {
            composSelYPipeline[p].PerFrameSetup();
        }

35      // print debug info
        //static bool first(true); if (first) { cout<<this<<endl; first=false; }
    } // PerFrameSetup

40      /////////////////////////////////
    // local computation functions

    void ComposSelYStage::RunForOneClockCycle()
    {
        // communication
45        // computation
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            composSelYPipeline[p].RunForOneClockCycle();
        }
    } // RunForOneClockCycle
50

```

```

5   //////////////////////////////////////////////////////////////////
// internal utility functions

10  // end of ComposSelYStage.C
:::cube4/ComposSelYStage.h
:::ComposSelYStage.h
// (c) Ingmar Bitter '97

15  // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#define _ComposSelYStage_h_ // prevent multiple includes
20  #include "Misc.h"
#include "Object.h"
#include "Vector3D.h"
#include "Coxel.h"
#include "Control.h"
25  #include "ComposSelYPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

30  class Cube4;

35  class ComposSelYStageInputs {
public: // pointers
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

40  class ComposSelYStageResults {
public: // arrays
    Coxel *coixel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

45  class ComposSelYStage : virtual public Object {
public:
    static void Demo ();
};

55

```

72

```

// constructors & destructors
ComposSelYStage ();
~ComposSelYStage ();

// show/set data & data properties
// - class Object requirements
virtual ostream & Ostream (ostream &) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
                        const int setNumOfPipelinesPerChip,
                        const int setBlockSize,
                        Cube4 *setCube4);
virtual void LocalSetup (const int setChipIndex);
virtual void PerFrameSetup ();
// local computation functions
virtual void RunForOneClockCycle();

public:
    ComposSelYPipeline *composSelYPipeline;
    ComposSelYStageInputs inputs;
    ComposSelYStageResults results;

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    int          chipIndex; // only for debugging purpose

    friend class ComposSelYPipeline;
};

#endif // _ComposSelYStage_h_
:::::::::::
cube4/Coxel.C
:::::::::::
// Coxel.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include"Coxel.h"

Coxel::Coxel(FixPointNumber setR, FixPointNumber setG, FixPointNumber setB,
             FixPointNumber setA)
    : r(setR),g(setG),b(setB),a(setA)
{
} // constructor

```

50

55

```

73
Coxel::Coxel(const Coxel & coxel)
    : r(coxel.r),g(coxel.g),b(coxel.b),a(coxel.a)
5
} // constructor

Coxel::Coxel(const double & equalSet)
    : r(equalSet),g(equalSet),b(equalSet),a(equalSet)
10
{
} // constructor

// friend output function
ostream & operator << (ostream & os, const Coxel & c)
{ return os<<"("<<c.r<<","<<c.g<<","<<c.b<<","<<c.a<<")"; }
:::::::::::
cube4/ComposStage.C
:::::::::::
// ComposStage.C
20
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// Premultiplied Colors (C implies A*C)
25
//
// Back-to-Front
// Cacc = (1-Anew)Cacc + Cnew
// Aacc = (1-Anew)Aacc + Anew
//
30
// Front-to-Back
// Cacc = (1-Aacc)Cnew + Cacc
// Aacc = (1-Aacc)Anew + Aacc

#include "ComposStage.h"

35
void ComposStage::Demo()
{
    ComposStage compos;
    cout << endl <<"Demo of class " << typeid(compos).name();
    cout << endl <<"size : " << sizeof(ComposStage) << " Bytes";
    cout << endl <<"public member functions:";
40
    cout << endl <<"ComposStage compos; = " << compos;
    cout << endl << "End of demo of class "<< typeid(compos).name() << endl;
} // Demo

45
///////////////////////////////
// constructors & destructors

// static first init
int ComposStage::numOfChips           = 0;
int ComposStage::numOfPipelinesPerChip = 0;
50

```

74

```

Cube4 *ComposStage::cube4 = 0;

5   ComposStage::ComposStage()
{
    composPipeline = new ComposPipeline [numOfPipelinesPerChip];
    tempResults.coxel = new Coxel [numOfPipelinesPerChip];
    tempResults.perPipelineControlFlags = new PerPipelineControlFlags
[numOfPipelinesPerChip];
10  results.coxel = new Coxel [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
[numOfPipelinesPerChip];
} // defaultconstructor

15  ComposStage::~ComposStage()
{
    if (composPipeline) { delete composPipeline; composPipeline=0; }
    if (tempResults.coxel) { delete tempResults.coxel; tempResults.coxel=0; }
    if (tempResults.perPipelineControlFlags) {
        delete tempResults.perPipelineControlFlags;
20      tempResults.perPipelineControlFlags=0;
    }
    if (results.coxel) { delete results.coxel; results.coxel=0; }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
25      results.perPipelineControlFlags=0;
    }
} // destructor

30  //////////////////////////////////////////////////////////////////
// show/set data & data properties

ostream & ComposStage::Ostream(ostream & os) const
{
    // append ComposStage info to os
35  os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex           = " << chipIndex;

40  // return complete os
    return os;
}

45  //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

50

```

55

```

void ComposStage::GlobalSetup(const int setNumOfChips,
5
    const int setNumOfPipelinesPerChip,
    Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
10   numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    cube4              = setCube4;
    ComposPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip);
} // GlobalSetup

15 void ComposStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composPipeline[p].LocalSetup(chipIndex,p,*this);
    }
20 } // LocalSetup

void ComposStage::PerFrameSetup()
{
25   int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.coxel[p] = cube4->backgroundCoxel;
        results.perPipelineControlFlags[p].Reset();
    }
30   results.perChipControlFlags.Reset();

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        composPipeline[p].PerFrameSetup();
    }

35   // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

40 ///////////////////////////////// local computation functions

void ComposStage::RunForOneClockCycle()
{
45   for (int p=0; p<numOfPipelinesPerChip; ++p) {
        composPipeline[p].RunForOneClockCycle();
        tempResults.perPipelineControlFlags[p] =
inputs.perPipelineControlFlags[p];
    }
        tempResults.perChipControlFlags = *(inputs.perChipControlFlags);
50
}

```

```

    } // RunForOneClockCycle

5   void ComposStage::WriteResultsToPipelineRegister()
{
    results.perChipControlFlags = tempResults.perChipControlFlags;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.coxel[p] = tempResults.coxel[p];
        results.perPipelineControlFlags[p] =
            tempResults.perPipelineControlFlags[p];
    }
} // WriteResultsToPipelineRegister

15
///////////////////////////////
// internal utility functions

20 // end of ComposStage.C
:::::::::::
cube4/ComposStage.h
:::::::::::
// ComposStage.h
// (c) Ingmar Bitter '97
25
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// Premultiplied Colors (C implies A*C)
30
// Back-to-Front
// Cacc = (1-Anew)Cacc + Cnew
// Aacc = (1-Anew)Aacc + Anew
//
35
// Front-to-Back
// Cacc = (1-Aacc)Cnew + Cacc
// Aacc = (1-Aacc)Anew + Aacc

40 #ifndef _ComposStage_h_ // prevent multiple includes
#define _ComposStage_h_

#include "Misc.h"
#include "Object.h"
#include "Voxel.h"
45 #include "Coxel.h"
#include "Control.h"
#include "ComposPipeline.h"

50 class ComposStageInputs {
public: // pointers
    Shadel *shadel;
    Coxel *coixel;
}

```

55

```

      77
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
  5   };

  class ComposStageResults {
10  public: // arrays
    Coxel *coixel;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
  };

15  class ComposStage : virtual public Object {
public:

  static void Demo ();

20  // constructors & destructors
  ComposStage ();
  ~ComposStage ();

25  // show/set data & data properties
  // - class Object requirements
  virtual ostream & Ostream (ostream &) const;

  // - local show/set functions
30  static void GlobalSetup (const int setNumberOfChips,
                           const int setNumberOfPipelinesPerChip,
                           Cube4 *setCube4);
  virtual void LocalSetup (const int setChipIndex);
  virtual void PerFrameSetup ();
35  // local computation functions
  virtual void RunForOneClockCycle();
  virtual void WriteResultsToPipelineRegister();

40  public:
    ComposPipeline *composPipeline;
    ComposStageInputs inputs;
    ComposStageResults tempResults;
    ComposStageResults results;

45  protected:
    static int numofChips, numofPipelinesPerChip;
    static Cube4 *cube4;
    int chipIndex; // only for debugging purpose

50  friend class ComposPipeline;
};

55

```

```

5      #endif // _ComposStage_h_
:::::::::::
cube4/Control.C
:::::::::::
// Control.C
// (c) Ingmar Bitter '97

10     // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15     #include "Control.h"
#include "Cube4.h"

void Control::Demo()
{
    Control control;
    cout << endl << "Demo of class " << typeid(control).name();
20     cout << endl << "size : " << sizeof(Control) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "Control control; = " << control;
    cout << endl << "End of demo of class " << typeid(control).name() << endl;
} // Demo

25     PerChipControlFlags::PerChipControlFlags()
{
    Reset();
} // constructor

30     PerChipControlFlags::PerChipControlFlags(PerChipControlFlags & src)
: volumeStart(src.volumeStart),
sliceSlabStart(src.sliceSlabStart),
beamSlabStart(src.beamSlabStart),
35     xBlockStart(src.xBlockStart),
xBlockEnd(src.xBlockEnd),
yBlockStart(src.yBlockStart),
zBlockStart(src.zBlockStart),
xStep(src.xStep),
40     yStep(src.yStep),
endFace(src.endFace),
leftmostChip(src.leftmostChip),
rightmostChip(src.rightmostChip)
{
} // copy constructor

45     void PerChipControlFlags::Reset()
{
    50     volumeStart      = false;
    sliceSlabStart   = false;
    beamSlabStart   = false;
}

```

79

```

5      xBlockStart      = false;
xBlockEnd       = false;
yBlockStart      = false;
zBlockStart      = false;
xStep           = Middle;
yStep           = Middle;
endFace         = None;
leftmostChip    = false;
rightmostChip   = false;
10     } // Reset

15     ostream & PerChipControlFlags::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << "@" << (void *) this;

    // return complete os
    return os;
20     } // Ostream

25     PerPipelineControlFlags::PerPipelineControlFlags()
{
    Reset();
} // constructor

30     PerPipelineControlFlags::PerPipelineControlFlags(PerPipelineControlFlags & src)
    : valid(src.valid), startOfRay(src.startOfRay), endOfRay(src.endOfRay),
    voxelPosXYZ(src.voxelPosXYZ)
{
} // constructor

35     void PerPipelineControlFlags::Reset()
{
    valid = false;
    startOfRay = false;
    endOfRay = false;
    voxelPosXYZ(0,0,0);
} // Reset
40

45     ostream & PerPipelineControlFlags::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << "@" << (void *) this;

    // return complete os
    return os;
} // Ostream

```

40

55

```

////////// // class Control implementation

5   // static first init
    int Control::numOfChips          = 0;
    int Control::numOfPipelinesPerChip = 0;
    int Control::blockSize          = 0;
10   Cube4 *Control::cube4           = 0;

////////// // constructors & destructors

15  Control::Control()
{
    results.voxelPosXYZ = new Vector3D<int> [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
[numOfPipelinesPerChip];
20   } // constructor

Control::~Control()
{
25   if (results.voxelPosXYZ) {
        delete results.voxelPosXYZ;
        results.voxelPosXYZ=0;
    }
    if (results.weightsXYZ) {
        delete results.weightsXYZ;
      results.weightsXYZ=0;
    }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
35   } // destructor

////////// // show/set data & data properties

40

ostream & Control::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << "@" << (void *) this;
45   os << endl << "  numOfChips      = " << numOfChips;
    os << endl << "  chipIndex       = " << chipIndex;
    os << endl << "  datasetSizeXYZ = " << datasetSizeXYZ;

    // return complete os
50   return os;
}

```

```

} // Ostream
5

///////////////////////////////
// show/set data & data properties
//
// - local show/set functions
10

void Control::GlobalSetup(const int setNumOfChips,
                          const int setNumOfPipelinesPerChip,
                          const int setBlockSize,
                          Cube4 *setCube4)
15
{
    numOfChips      = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize       = setBlockSize;
    cube4 = setCube4;
20 } // GlobalSetup

void Control::LocalSetup(const int setChipIndex)
{
25     chipIndex      = setChipIndex;
} // LocalSetup

void Control::PerFrameSetup(const Vector3D<int> & setDatasetSizeXYZ,
                           const Vector3D<int> startVoxelPos,
                           const Vector3D<FixPointNumber> & setViewPointXYZ,
                           const CompositingStyles & setCompositingStyle,
                           const ProjectionStyles & setProjectionStyle,
                           const CubeModes & setCubeMode)
30
{
    int X,Y,Z;
    datasetSizeXYZ = setDatasetSizeXYZ;
    X = setDatasetSizeXYZ.X();
    Y = setDatasetSizeXYZ.Y();
    Z = setDatasetSizeXYZ.Z();
    voxelPos( ModInt(startVoxelPos.X(), X),
40             ModInt(startVoxelPos.Y(), Y),
             ModInt(startVoxelPos.Z(), Z));
    leftRightDistance = 1;
    viewPointXYZ = setViewPointXYZ;
    normalizedSightRay(0,0,0);
    samplePosIncrement(0,0,0);
45    samplePosXYZ(0,0,0);
    compositingStyle = setCompositingStyle;
    projectionStyle = setProjectionStyle;
    cubeMode = setCubeMode;
    leftmostChipIndex(-2,numOfChips); // -2 because there will be 2
50    ++leftmostChipIndex

```

```

82
rightmostChipIndex(-3,numOfChips); // while computing the first IntVoxelPos()

5      zDepth = 4;

x_voxelStep(ModInt(1,X),ModInt(0,Y),ModInt(0,Z));
y_voxelStep(ModInt(0,X),ModInt(1,Y),ModInt(0,Z));
z_voxelStep(ModInt(0,X),ModInt(-blockSize,Y),ModInt(1,Z)); // includes y
corection
10     x_blockStep(ModInt(blockSize*(numOfChips-1),X),ModInt(0,Y),ModInt(0,Z)); //
compute this is on each chip
      y_blockStep(ModInt(-blockSize,X),ModInt(blockSize,Y),ModInt(-blockSize,Z)); //
consider skewing and z-correction
      z_blockStep(ModInt(-blockSize,X),ModInt(0,Y),ModInt(blockSize,Z)); // consider
skewing
15
x_partialBlockBeamStep(ModInt(blockSize*(numOfChips),X),ModInt(0,Y),ModInt(0,Z))
;

// reset pipeline registers
20   for (int p=0; p<numOfPipelinesPerChip; ++p) {
      results.voxelPosXYZ[p](0,0,0);
      results.perPipelineControlFlags[p].Reset();
    }
  results.perChipControlFlags.Reset();

25   if (projectionStyle == parallel) {
      sightRay = datasetSizeXYZ;
      sightRay /= 2.0;
      sightRay -= viewPointXYZ;

      // normalize by z component
30   FixPointNumber norm(sightRay.Z().Abs());
      normalizedSightRay(sightRay.X() / norm,
                          sightRay.Y() / norm,
                          0);

      if (compositingStyle == FrontToBack)
        samplePosIncrement = normalizedSightRay;
      else if (compositingStyle == BackToFront)
        samplePosIncrement = -normalizedSightRay;

      if (chipIndex ==0)
        cout << "normalizedSightRay:" << normalizedSightRay
          << "samplePosIncrement:" << samplePosIncrement << endl;
    }

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
45 } // PerFrameSetup

///////////////////////////////
// local computation functions

```

50

55

```

void Control::RunForOneClockCycle()
{
    // walk left to right within beams
    //   top   bottom   slices
    //   front back   volume cube

    // do this for whole blocks and within blocks
    // init flags to defaults within volume
    results.perChipControlFlags.Reset();
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.perPipelineControlFlags[p].Reset();
    }

    // compute integer voxel read positions (in pipeline xyz coords)
    IntVoxelPos();

    // compute weights for TriLin associated with voxel pos
    TriLinWeights();
}

// RunForOneClockCycle

void Control::IntVoxelPos()
{
    /////////////////////////////////
    // compute integer voxel read positions (in pipeline xyz coords)

    int p;
    static int volNum(-1);
    // go to next voxel to the right
    voxelPos += x_voxelStep;

    if ((voxelPos.X() + numOfPipelinesPerChip * x_voxelStep.X()) % blockSize == 0) {
        // will run over block boundary in next cycle
        results.perChipControlFlags.xBlockEnd = true;
    }
    if (voxelPos.X() % blockSize == 0) {
        // just ran over block boundary,
        // need to go to next partial beam in next block
        voxelPos += x_blockStep;
        results.perChipControlFlags.xBlockStart = true;
    }
    if (voxelPos.X() % blockSize == 0 &&
        voxelPos.X() < blockSize * numChips) {
        // just ran out of volume, need to go to next beam of voxels
        voxelPos += y_voxelStep;
    }
    if (voxelPos.Y() % blockSize == 0) {
        // just ran over block boundary, need to go to next z-beam inside block
        voxelPos += z_voxelStep;
        results.perChipControlFlags.blockSliceStart = true;
    }
}

```

50

55

```

84
//if (voxelPos.Z() == datasetSizeXYZ.Z() - 3)
//results.perChipControlFlags.startReadingMiniblocks;
5
if (voxelPos.Z() % blockSize == 0) {
    // just ran too deep in z, need to come back and step one block in y
    voxelPos += y_blockStep;
    if (chipIndex == leftmostChipIndex)
        voxelPos += x_partialBlockBeamStep;
    ++leftmostChipIndex;
    ++rightmostChipIndex;
    results.perChipControlFlags.yBlockStart = true;
    if (chipIndex == 0) { cout<<"y"; cout.flush(); }

10
if (voxelPos.Y() == 0) {
    // just ran out of volume, need to go to next slice of blocks
    voxelPos += z_blockStep;
    if (chipIndex == leftmostChipIndex)
        voxelPos += x_partialBlockBeamStep;
    ++leftmostChipIndex;
    ++rightmostChipIndex;
    results.perChipControlFlags.volumeSlabStart = true;
    results.perChipControlFlags.zBlockStart = true;
    // debug info
    if (chipIndex == 0)      cout << "S";

20
if (voxelPos.Z() == 0) {
    // just ran out of volume, need to wrap back to volume start
    // assuming that skewing has made a full cycle and doesn't
    // need to be corrected this time
    results.perChipControlFlags.volumeStart = true;
    ++volNum;
25
}
}
}
}
}

30
}
}
}
}

35
}

// write voxel positions into pipeline registers
// now at first correct voxel
// following voxels are guaranteed to be within the same block
40
results.voxelPosXYZ[0] = voxelPos;
for (p=1; p<numOfPipelinesPerChip; ++p) {
    voxelPos += x_voxelStep;
    results.voxelPosXYZ[p] = voxelPos; // for debugging
    // make image generation easier by not wrapping the z coordinate
    results.voxelPosXYZ[p].SetZ(voxelPos.Z() +
45
volNum*datasetSizeXYZ.Z());
}

50
// send pos also as control
// only for debugging purpose
for (p=0; p<numOfPipelinesPerChip; ++p) {
}

```

```

5      results.perPipelineControlFlags[p].voxelPosXYZ = results.voxelPosXYZ[p];
       // in case one wants to see the voxel time stamps, uncomment next
line
       //results.perPipelineControlFlags[p].voxelPosXYZ.SetZ(cube4->clk+1);
}

10     results.perChipControlFlags.leftmostChip = (chipIndex == leftmostChipIndex);
     results.perChipControlFlags.rightmostChip = (chipIndex == rightmostChipIndex);
} // IntVoxelPos

void Control::TriLinWeights()
{
15     /////////////////////////////////
     // compute weights for TriLin associated with voxel pos

    if (cubeMode == Cube4Light) {

20         for (int p=0; p<numOfPipelinesPerChip; ++p) {

            if (projectionStyle == perspective) {
                samplePosXYZ = results.voxelPosXYZ[p];

                sightRay = samplePosXYZ - viewPointXYZ;

25                // normalize by z component
                FixPointNumber norm(sightRay.Z().Abs());
                normalizedSightRay(sightRay.X() / norm,
                                    sightRay.Y() / norm,
                                    0);

30                if (compositingStyle == FrontToBack)
                    samplePosIncrement = normalizedSightRay;
                else if (compositingStyle == BackToFront)
                    samplePosIncrement = -normalizedSightRay;
            }

35                results.weightsXYZ[p] = samplePosIncrement;
        }
        // set x/y steps (in perspective more xsteps might have to be reset later)
        // X/Y steps mean where rays go (eg, if xstep > 0, ray makes step to the
right)
40        // CHANGE ACCORDINGLY!!!!!
        if (samplePosIncrement.X() > 0) xStep = Left;
        else if (samplePosIncrement.X() < 0) xStep = Right;
        else
            xStep = Middle;
        if (samplePosIncrement.Y() > 0) yStep = Up;
        else if (samplePosIncrement.Y() < 0) yStep = Down;
45        else
            yStep = Middle;

        samplePosXYZ(0,0,0); // local pos of trilin weights used in endOfRay
computation
    } // cubeMode == Cube4Light
}

```

```

else if (cubeMode == Cube4Classic) {

5      if (results.perChipControlFlags.blockSliceStart) {
          // advance samplePos by samplePosIncrement at beginning
          // of every slice of voxels in a block
          samplePosXYZ += samplePosIncrement;

10     // at z-block step update blockStartSamplePos
     if (results.perChipControlFlags.zBlockStart) {
             blockStartSamplePos = samplePosXYZ;
     }

15     // at y-block step restore samplePos of previous
     blockStartSamplePos
     if (results.perChipControlFlags.yBlockStart) {
             samplePosXYZ = blockStartSamplePos;
     }

20     // at start of volume set initial samplePosXYZ
     if (results.perChipControlFlags.volumeStart) {
             samplePosXYZ(0,0,0);
             blockStartSamplePos(0,0,0);
     }

25     // set x/y-steps to defaults
     xStep = Middle;
     yStep = Middle;

     // reset x/y-steps if necessary
     // X/Y steps mean where rays go (eg, if xstep > 0, ray makes step to the
30     right)
     if (samplePosXYZ.X() >= 1) { samplePosXYZ += Vector3D<FixPointNumber>(-
1,0,0); xStep = Right; }
     if (samplePosXYZ.X() < 0) { samplePosXYZ += Vector3D<FixPointNumber>(
1,0,0); xStep = Left; }
35     if (samplePosXYZ.Y() >= 1) { samplePosXYZ += Vector3D<FixPointNumber>(0,-
1,0); yStep = Down; }
     if (samplePosXYZ.Y() < 0) { samplePosXYZ += Vector3D<FixPointNumber>(0,
1,0); yStep = Up; }

40     // assign same weight to all pipelines (parallel projection only)
     for (int p=0; p<numOfPipelinesPerChip; ++p) {
             results.weightsXYZ[p] = samplePosXYZ;
     }
     // assign same x/y-steps to all pipelines (parallel projection only)
     results.perChipControlFlags.xStep = xStep;
45     results.perChipControlFlags.yStep = yStep;
} // cubeMode == Cube4Classic

///////////

```

```

// set startOfRay bits

5   // preset with default
for (int p=0; p<numOfPipelinesPerChip; ++p) {
    results.perPipelineControlFlags[p].startOfRay = false;
}

10  // check for first (front) face
if (results.voxelPosXYZ[0].Z() == 0) {
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.perPipelineControlFlags[p].startOfRay = true;
    }
}

15  // check for top or bottom overflow
else if ( // check for top overflow
          results.voxelPosXYZ[0].Y() == 0
          && yStep == Down
          || // check for bottom overflow
          results.voxelPosXYZ[0].Y() == datasetSizeXYZ.Y()-1
          && yStep == Up ) {

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.perPipelineControlFlags[p].startOfRay = true;
    }
}

25  // check for left/right overflow
else {
    // check for left overflow
    //if (cubeMode == Cube4Light && projectionStyle == perspective) {
        //if      (results.weightsXYZ[0].X() > 0) xStep = Left;
        //else if (results.weightsXYZ[0].X() < 0) xStep = Right;
        //else                                xStep = Middle;

        //}
    35  if (results.voxelPosXYZ[0].X() == 0 && xStep == Right) {
        results.perPipelineControlFlags[0].startOfRay = true;
    }
    else {
        // check for right overflow
        //if (cubeMode == Cube4Light && projectionStyle == perspective) {
            //if      (results.weightsXYZ[numOfPipelinesPerChip-1].X() >
0) xStep = Left;
            //else if (results.weightsXYZ[numOfPipelinesPerChip-1].X() <
0) xStep = Right;
            //else
45  xStep = Middle;
        //}
        if (results.voxelPosXYZ[numOfPipelinesPerChip-1].X()
            == datasetSizeXYZ.X() - 1
            && xStep == Left) {
}
}
50

```

```

results.perPipelineControlFlags[numOfPipelinesPerChip-1].startOfRay = true;
5      }
    }
}

///////////////////////////////
// set endOfRay and endFace bits
10
// preset with default
results.perChipControlFlags.endFace = None;
for (p=0; p<numOfPipelinesPerChip; ++p) {
    results.perPipelineControlFlags[p].endOfRay = false;
}
15

// check for final (back) face
if (results.voxelPosXYZ[0].Z() == datasetSizeXYZ.Z()-1) {
    results.perChipControlFlags.endFace = BackFace;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.perPipelineControlFlags[p].endOfRay = true;
}
20
}

// check for top or bottom overflow
if ( // check for top overflow
25      results.voxelPosXYZ[0].Y() == 0
&& samplePosXYZ.Y() +samplePosIncrement.Y() < 0
|| // check for bottom overflow
      results.voxelPosXYZ[0].Y() == datasetSizeXYZ.Y()-1
&& ((cubeMode==Cube4Classic && (samplePosXYZ.Y()
30
            +samplePosIncrement.Y() >= 1)) ||
(cubeMode==Cube4Light &&
results.weightsXYZ[0].Y() > 0) ) {
    results.perChipControlFlags.endFace = TopBottomFace;
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.perPipelineControlFlags[p].endOfRay = true;
}
}
35

// check for left/right overflow
if ( // check for left overflow
40      results.voxelPosXYZ[0].X() == 0
&& ((cubeMode==Cube4Classic && (samplePosXYZ.X()
            + samplePosIncrement.X() < 0))
|| (cubeMode==Cube4Light &&
results.weightsXYZ[0].X() < 0) ) ) {
    results.perChipControlFlags.endFace
        = results.perChipControlFlags.endFace | SideFace;
    results.perPipelineControlFlags[0].endOfRay = true;
}
45
50

```

```

89
}
else if ( // check for right overflow
5      results.voxelPosXYZ[numOfPipelinesPerChip-1].X()
      == datasetSizeXYZ.X()-1
      && ((cubeMode==Cube4Classic && (samplePosXYZ.X()
      +samplePosIncrement.X() >= 1))
      ||
10     (cubeMode==Cube4Light &&
      results.weightsXYZ[numOfPipelinesPerChip-1].X() > 0) ) {
      results.perChipControlFlags.endFace
      = results.perChipControlFlags.endFace | SideFace;
      results.perPipelineControlFlags[numOfPipelinesPerChip-1].endOfRay
15     = true;
    }

///////////////////////////////
// valid flag computation, same for all cube modes
20 for (p=0; p<numOfPipelinesPerChip; ++p) {
    // set valid flag, if not at volume boundary
    if (results.voxelPosXYZ[p].X() > 0 &&
        results.voxelPosXYZ[p].Y() > 0 &&
        results.voxelPosXYZ[p].Z() > 0 &&
        results.voxelPosXYZ[p].X() < datasetSizeXYZ.X()-2 &&
25       results.voxelPosXYZ[p].Y() < datasetSizeXYZ.Y()-2 &&
       results.voxelPosXYZ[p].Z() < datasetSizeXYZ.Z()-2 ) {
        results.perPipelineControlFlags[p].valid = true;
    }
    // set valid flag to false, if weights are suggesting non local
30   access
    else if (normalizedSightRay.X().Abs() > 1 || 
              normalizedSightRay.Y().Abs() > 1) {
        results.perPipelineControlFlags[p].valid = false;
        cout << "x", cout.flush();
    }
35 }
} // TriLinWeights

// end of Control.C
:::::::::::
cube4/Control.h
40 :::::::::::::
// Control.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.
45 #ifndef _Control_h_      // prevent multiple includes
#define _Control_h_

```

50

55

90

```

#include "Misc.h"
#include "Object.h"
#include "ModInt.h"
5 #include "FixPointNumber.h"
#include "Vector3D.h"
#include "Matrix4x4.h"

class Cube4;

10 class PerChipControlFlags {
public:
    bool volumeStart; // need bool, because SliceVoxelFiFo wants pointer to it
    unsigned sliceSlabStart : 1;
    unsigned beamSlabStart : 1;
15    unsigned xBlockStart : 1;
    unsigned xBlockEnd : 1;
    unsigned yBlockStart : 1;
    unsigned zBlockStart : 1;
    unsigned xStep : 2;
20    unsigned yStep : 2;
    unsigned endFace : 3;
    unsigned leftmostChip : 1;
    unsigned rightmostChip : 1;

    unsigned blockSliceStart : 1; // kill
25    unsigned volumeSlabStart : 1;
    unsigned blockStart : 1;
public:
    PerChipControlFlags();
    PerChipControlFlags(PerChipControlFlags & src);
    //void PerFrameSetup();
30    void Reset();
    virtual ostream & Ostream (ostream & ) const;
}; // PerChipControlFlags

class PerPipelineControlFlags {
public:
35    unsigned valid : 1;
    unsigned startOfRay : 1;
    unsigned endOfRay : 1;
    Vector3D<int> voxelPosXYZ; // only for debugging purpose
public:
    PerPipelineControlFlags();
40    PerPipelineControlFlags(PerPipelineControlFlags & src);
    // void PerFrameSetup();
    void Reset();
    virtual ostream & Ostream (ostream & ) const;
}; // PerPipelineControlFlags

45 class ControlResults {
public: // arrays
    Vector3D<int> *voxelPosXYZ;
    Vector3D<FixPointNumber> *weightsXYZ;
}

```

50

55

```

91
PerChipControlFlags           perChipControlFlags;
PerPipelineControlFlags *perPipelineControlFlags;
5 } // ControlResults

class Control : virtual public Object {
public:

10    static void      Demo ();

        // constructors & destructors
        Control ();
        ~Control ();

15    // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream & ) const;

20    // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,
                                const int setNumOfPipelinesPerChip,
                                const int setBlockSize,
                                Cube4 *cube4);
        virtual void LocalSetup(const int setChipIndex);
        virtual void PerFrameSetup(const Vector3D<int> & setDatasetSizeXYZ,
                                const Vector3D<int> startVoxelPos,
                                const Vector3D<FixPointNumber> & setViewPointXYZ,
                                const CompositingStyles & setCompositingStyle,
35                                const ProjectionStyles & setProjectionStyle,
                                const CubeModes & setCubeMode);
        // local computation functions
40        virtual void RunForOneClockCycle();
        virtual void IntVoxelPos();
        virtual void TriLinWeights();

public:
45    ControlResults results;

protected:
        static int      numOfChips, numOfPipelinesPerChip, blockSize;
        int             chipIndex;
        ModInt          leftmostChipIndex, rightmostChipIndex;
50        Vector3D<int> datasetSizeXYZ; // pipeline coordinates
        Vector3D<ModInt> voxelPos;
        Vector3D<ModInt> x_voxelStep,y_voxelStep,z_voxelStep;

```

55

```

92
Vector3D<ModInt>           x_blockStep,y_blockStep,z_blockStep;
Vector3D<ModInt> x_partialBlockBeamStep;
Vector3D<FixPointNumber> viewPointXYZ;
5   Vector3D<FixPointNumber> samplePosXYZ, blockStartSamplePos;
Vector3D<FixPointNumber> sightRay;
Vector3D<FixPointNumber> normalizedSightRay;
Vector3D<FixPointNumber> samplePosIncrement;
int xStep, yStep;
10  int leftRightDistance, zDepth;
static Cube4 *cube4;

CompositingStyles compositingStyle;
ProjectionStyles projectionStyle;
15  CubeModes      cubeMode;

friend class Cube4;
};

#endif      // _Control_h_
20 ::::::::::::::
cube4/CoxMem.C
::::::::::::::::::
// CoxMem.C
// (c) Ingmar Bitter '97

25 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// DRAM Coxel Memory of one pipeline
// in : 1/numOfPipeline fraction of the final image
30 // out: random access to any of these coixels

#include "CoxMem.h"

void CoxMem::Demo()
{
35     CoxMem mem;
    cout << endl << "Demo of class " << typeid(mem).name();
    cout << endl << "size : " << sizeof(CoxMem) << " Bytes";
    cout << endl << "public member functions:";
    mem(3)=65; cout << endl << " mem(3)=65      = " << mem(3);
40     cout << endl << "CoxMem::Demo() end" << endl;
} // Demo

///////////////////////////////
// constructors & destructors
45 CoxMem::CoxMem()
    : size(0), mem(0)
{
} // constructor
50

```

```

5      CoxMem::~CoxMem()
6      {
7          if (mem) { delete mem; mem=0; }
8      } // destructor

10     //////////////////////////////////////////////////////////////////
11     // show/set data & data properties

12     ostream & CoxMem::Ostream(ostream & os) const
13     {
14         // append Control info to os
15         os << typeid(*this).name() << "@" << (void *) this;
16         os << endl << " size = " << size;

17         // return complete os
18         return os;
19     } // Ostream

20     //////////////////////////////////////////////////////////////////
21     // - local show/set functions

22     bool CoxMem::SetSize (unsigned int setSize)
23     {
24         size = setSize;
25         if (mem) delete mem;
26         mem = new Coxel [size];
27         if (!mem) ERROR("OutOfMemoryError");
28         return true;
29     } // SetSize

30     Coxel & CoxMem::operator () (unsigned int index)
31     {
32         // subscript a array value
33         // check that index is valid
34         if (index >= size) {
35             cout << index << "index:size" << size;
36             ERROR("TooLargeArrayIndexError!");
37         }

38         // return reference to requested element
39         return mem[index];
40     } // operator ()

41     // end of CoxMem.C
42     ::::::::::::::
43     cube4/CoxMem.h
44     ::::::::::::::

```

94

```

// CoxMem.h
// (c) Ingmar Bitter '97
5
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// DRAM Coxel Memory of one pipeline
10 // in : 1/numOfChips fraction of the final image
// out: bustmode access to blocks of voxels

#ifndef _CoxMem_h_           // prevent multiple includes
#define _CoxMem_h_
15

#include "Object.h"
#include "Misc.h"
#include "Coxel.h"

20 class CoxMem : virtual public Object {
public:

    static void      Demo ();

    // constructors & destructors
25 CoxMem();
    virtual ~CoxMem();

    // show/set data & data properties
    // - class Object requirements
30     virtual       ostream &   Ostream (ostream &) const;

    virtual         bool      SetSize    (unsigned int setSize);
    virtual /*inline*/ Coxel & operator () (unsigned int index);

35     // local computation functions

protected:
    Coxel *mem;
    unsigned int size;
40};

#endif      // _CoxMem_h_
:::::::::::
cube4/Coxel.h
:::::::::::
// Coxel.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
50 // America, Inc., 1997, All rights reserved.

#ifndef _Coxel_h_ // prevent multiple includes
#define _Coxel_h_

```

55

95

```

5      #include <iostream.h> // cout
6      #include "FixPointNumber.h"
7
8      class Coxel {
9      public: // data
10         FixPointNumber r,g,b,a;
11     public: // member functions
12         Coxel(FixPointNumber r=0, FixPointNumber g=0, FixPointNumber b=0,
13               FixPointNumber a=0);
14         Coxel(const Coxel & coxel);
15         Coxel(const double & equalSet);
16     public: // friend functions
17         friend ostream & operator << (ostream & os, const Coxel & c);
18     }; // Coxel
19
20 #endif      // _Coxel_h_
21 ::::::::::::::
22 cube4/Cube4.C
23 ::::::::::::::
24 // Cube4.C
25 // (c) Ingmar Bitter '97 / Urs Kanus '97
26
27 // Copyright, Mitsubishi Electric Information Technology Center
28 // America, Inc., 1997, All rights reserved.
29
30 // Cube4 complete parallel rendering pipeline
31 // in : dataset, colorTable, illumTable
32 // out: baseplane image
33
34 //::::::::::::::::::
35 // constructors & destructors
36
37 Cube4::Cube4(int argc, char *argv[])
38     : cubeMode(Cube4Classic), projectionStyle(parallel),
39       numOfChips(4), numOfPipelinesPerChip(4), blockSize(8),
40       numOfFramesInAnimation(0),
41       trilinWeightBits(8), gradientBits(8), .
42       shaderMode(Phong),
43       colorTableLoaded(false), alphaTableLoaded(false),
44       reflectanceTableLoaded(false),
45       viewPointSet(false), phongExponent(50),
46       IAmbient(1.0, 1.0, 1.0),
47       Ka(0.6), Kd(0.5), Ks(1.0),
48       threshold(0.6), maxWidth(1.5), mulAlpha(2.0),
49       shadelColorBits(8), shadelAlphaBits(8),
50       compositingStyle(BackToFront),
51       coxelColorBits(8), coxelAlphaBits(8),
52       singleImagePerFrameMode(false),
53       distortionCompensation(true),
54
55

```

```

96
    constantAngularResolution(true),
reflectanceMapIP(true),
reflectanceMapBits(3),
mapWeightBits(3)
{
    int c;

    strcpy(datasetFileName,"bulb.slc");
    strcpy(imageRootFileName,"Image");
    strcpy(colorTableFileName,"RGBA-LookupTable");
    alphaTableFileName[0] = 0;
    reflectanceTableFileName[0] = 0;

    ReadCube4PipelineParametersFromFile("Cube4PipelineParameter.dat");
    ReadCube4PipelineParametersFromCommandLine(argc, argv);

    if (cubeMode == Cube4Classic && projectionStyle == perspective) {
        cout << "\n sorry Cube4Classic & perspective not yet possible \n";
    exit(1);
}

///////////////////////////////
// allocate memory for pipeline objects

voxMem = new VoxMem [numOfChips];
coxMem = new CoxMem [numOfChips];

Control::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
AddressGenerator::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize);
MemoryCtrl::GlobalSetup      (numOfChips,numOfPipelinesPerChip);
SliceVoxelFiFoStage::GlobalSetup(numOfChips,numOfPipelinesPerChip,blockSize,this);
TriLinZStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
TriLinYStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
TriLinXStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
GradYStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
GradXStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
GradZStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
GradZLinZStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
GradZLinYStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
GradZLinXStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
ShaderStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
DataSyncStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);
ComposBufferStage::GlobalSetup
(numOfChips,numOfPipelinesPerChip,blockSize,this);

```

50

55

```

    ComposStage::GlobalSetup
    (numOfChips,numOfPipelinesPerChip,this);
5     ComposSelXStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
     ComposSelYStage::GlobalSetup
    (numOfChips,numOfPipelinesPerChip,blockSize,this);
     ComposLinXStage::GlobalSetup      (numOfChips,numOfPipelinesPerChip,this);
     ComposLinYStage::GlobalSetup
7    (numOfChips,numOfPipelinesPerChip,blockSize,this);
     FinalCoxelBuffer::GlobalSetup
    (numOfChips,numOfPipelinesPerChip,blockSize,this);

    control      = new Control          [numOfChips];
    addrGen       = new AddressGenerator [numOfChips];
15   memCtrl       = new MemoryCtrl     [numOfChips];
    sliceVoxelFiFo0 = new SliceVoxelFiFoStage[numOfChips];
    sliceVoxelFiFo1 = new SliceVoxelFiFoStage[numOfChips];
    triLinZ       = new TriLinZStage    [numOfChips];
    triLinY       = new TriLinYStage    [numOfChips];
    triLinX       = new TriLinXStage    [numOfChips];
20   gradY         = new GradyStage     [numOfChips];
    gradX         = new GradXStage     [numOfChips];
    gradZ         = new GradZStage     [numOfChips];
    gradZLinZ     = new GradZLinZStage [numOfChips];
    gradZLinY     = new GradZLinYStage [numOfChips];
25   gradZLinX     = new GradZLinXStage [numOfChips];
    shader        = new ShaderStage     [numOfChips];
    dataSync       = new DataSyncStage  [numOfChips];
    composBuff     = new ComposBufferStage[numOfChips];
    compos         = new ComposStage    [numOfChips];
    composSelX     = new ComposSelXStage [numOfChips];
    composSelY     = new ComposSelYStage [numOfChips];
30   composLinX     = new ComposLinXStage [numOfChips];
    composLinY     = new ComposLinYStage [numOfChips];
    finalCoxelBuff = new FinalCoxelBuffer [numOfChips];

35   /////////////////////////////////
// setup pipeline connections

for (c=0; c<numOfChips; ++c) {
    control[c].LocalSetup(c);
}
40
for (c=0; c<numOfChips; ++c) {
    addrGen[c].inputs.voxelPosXYZ = control[c].results.voxelPosXYZ;
    addrGen[c].inputs.weightsXYZ = control[c].results.weightsXYZ;
    addrGen[c].inputs.perChipControlFlags
45   = &(control[c].results.perChipControlFlags);
    addrGen[c].inputs.perPipelineControlFlags
        = control[c].results.perPipelineControlFlags;
}

for (c=0; c<numOfChips; ++c) {
50   memCtrl[c].inputs.address = addrGen[c].results.address;
}

```

```

98
memCtrl[c].inputs.weightsXYZ =
addrGen[c].results.weightsXYZ;
5      memCtrl[c].inputs.perChipControlFlags
      = &(addrGen[c].results.perChipControlFlags);
memCtrl[c].inputs.perPipelineControlFlags
= addrGen[c].results.perPipelineControlFlags;
memCtrl[c].LocalSetup(c,&(voxMem[c]));
}
10
for (c=0; c<numOfChips; ++c) {
    sliceVoxelFiFo0[c].inputs.voxel = memCtrl[c].results.voxel;
    sliceVoxelFiFo0[c].inputs.weightsXYZ =
memCtrl[c].results.weightsXYZ;
    sliceVoxelFiFo0[c].inputs.perChipControlFlags
15      = &(memCtrl[c].results.perChipControlFlags);
    sliceVoxelFiFo0[c].inputs.perPipelineControlFlags
      = memCtrl[c].results.perPipelineControlFlags;
    sliceVoxelFiFo0[c].startReadWriteCounters
      = &(memCtrl[c].results.perChipControlFlags.volumeStart);
    sliceVoxelFiFo0[c].LocalSetup(c, sliceVoxelFiFo0);
}
20
for (c=0; c<numOfChips; ++c) {
    sliceVoxelFiFo1[c].inputs.voxel = sliceVoxelFiFo0[c].results.voxel;
    sliceVoxelFiFo1[c].inputs.weightsXYZ =
sliceVoxelFiFo0[c].results.weightsXYZ;
    sliceVoxelFiFo1[c].inputs.perChipControlFlags
25      = &(sliceVoxelFiFo0[c].results.perChipControlFlags);
    sliceVoxelFiFo1[c].inputs.perPipelineControlFlags
      = sliceVoxelFiFo0[c].results.perPipelineControlFlags;
    sliceVoxelFiFo1[c].startReadWriteCounters
      =
&(memCtrl[c].results.delayedPerChipControlFlags.volumeStart);
    sliceVoxelFiFo1[c].LocalSetup(c, sliceVoxelFiFo1);
}
30
for (c=0; c<numOfChips; ++c) {
    triLinZ[c].inputs.voxel0 = sliceVoxelFiFo0[c].results.voxel;
    triLinZ[c].inputs.voxel1 = sliceVoxelFiFo1[c].results.voxel;
    triLinZ[c].inputs.weightsXYZ =
sliceVoxelFiFo1[c].results.weightsXYZ;
    triLinZ[c].inputs.perChipControlFlags
35      = &(sliceVoxelFiFo1[c].results.perChipControlFlags);
    triLinZ[c].inputs.perPipelineControlFlags
      = sliceVoxelFiFo1[c].results.perPipelineControlFlags;
    triLinZ[c].LocalSetup(c);
}
40
for (c=0; c<numOfChips; ++c) {
    triLinY[c].inputs.voxel = triLinZ[c].results.voxel;
    triLinY[c].inputs.weightsXYZ = triLinZ[c].results.weightsXYZ;
    triLinY[c].inputs.perChipControlFlags
      = &(triLinZ[c].results.perChipControlFlags);
}
45
50

```

99

```

5      triLinY[c].inputs.perPipelineControlFlags
          = triLinZ[c].results.perPipelineControlFlags;
     triLinY[c].LocalSetup(c);
}

10     for (c=0; c<numOfChips; ++c) {
     triLinX[c].inputs.voxel = triLinY[c].results.voxel;
     triLinX[c].inputs.weightsXYZ = triLinY[c].results.weightsXYZ;
     triLinX[c].inputs.perChipControlFlags
          = &(triLinY[c].results.perChipControlFlags);
     triLinX[c].inputs.perPipelineControlFlags
          = triLinY[c].results.perPipelineControlFlags;
15     triLinX[c].LocalSetup(c);
}

20     for (c=0; c<numOfChips; ++c) {
     gradY[c].inputs.voxel = triLinX[c].results.voxel;
     gradY[c].inputs.weightsXYZ = triLinX[c].results.weightsXYZ;
     gradY[c].inputs.perChipControlFlags
          = &(triLinX[c].results.perChipControlFlags);
     gradY[c].inputs.perPipelineControlFlags
          = triLinX[c].results.perPipelineControlFlags;
25     gradY[c].LocalSetup(c);
}

30     for (c=0; c<numOfChips; ++c) {
     gradX[c].inputs.voxel = gradY[c].results.voxel;
     gradX[c].inputs.gy = gradY[c].results.gy;
     gradX[c].inputs.weightsXYZ = gradY[c].results.weightsXYZ;
     gradX[c].inputs.perChipControlFlags
          = &(gradY[c].results.perChipControlFlags);
     gradX[c].inputs.perPipelineControlFlags
          = gradY[c].results.perPipelineControlFlags;
35     gradX[c].LocalSetup(c);
}

40     for (c=0; c<numOfChips; ++c) {
     gradZ[c].inputs.voxel0 = memCtrl[c].results.voxel;
     gradZ[c].inputs.voxel1 = sliceVoxelFiFo1[c].results.voxel;
     gradZ[c].inputs.weightsXYZ = sliceVoxelFiFo0[c].results.weightsXYZ;
     gradZ[c].inputs.perChipControlFlags
          = &(sliceVoxelFiFo0[c].results.perChipControlFlags);
     gradZ[c].inputs.perPipelineControlFlags
          = sliceVoxelFiFo0[c].results.perPipelineControlFlags;
45     gradZ[c].LocalSetup(c);
}

50     for (c=0; c<numOfChips; ++c) {
     gradZLinZ[c].inputs.gz = gradZ[c].results.gz;
     gradZLinZ[c].inputs.weightsXYZ = gradZ[c].results.weightsXYZ;
     gradZLinZ[c].inputs.perChipControlFlags
          = &(gradZ[c].results.perChipControlFlags);
}

```

55

100

```

gradZLinZ[c].inputs.perPipelineControlFlags
    = gradZ[c].results.perPipelineControlFlags;
gradZLinZ[c].LocalSetup(c);
}

for (c=0; c<numOfChips; ++c) {
    gradZLinY[c].inputs.gz = gradZLinZ[c].results.gz;
    gradZLinY[c].inputs.weightsXYZ = gradZLinZ[c].results.weightsXYZ;
    gradZLinY[c].inputs.perChipControlFlags
        = &(gradZLinZ[c].results.perChipControlFlags);
    gradZLinY[c].inputs.perPipelineControlFlags
        = gradZLinZ[c].results.perPipelineControlFlags;
gradZLinY[c].LocalSetup(c);
}

for (c=0; c<numOfChips; ++c) {
    gradZLinX[c].inputs.gz = gradZLinY[c].results.gz;
    gradZLinX[c].inputs.weightsXYZ = gradZLinY[c].results.weightsXYZ;
    gradZLinX[c].inputs.perChipControlFlags
        = &(gradZLinY[c].results.perChipControlFlags);
    gradZLinX[c].inputs.perPipelineControlFlags
        = gradZLinY[c].results.perPipelineControlFlags;
gradZLinX[c].LocalSetup(c);
}

/* 
for (c=0; c<numOfChips; ++c) {
    shader[c].inputs.voxel = gradX[c].results.voxel;
    shader[c].inputs.gx = gradX[c].results.gx;
    shader[c].inputs gy = gradX[c].results.gy;
    shader[c].inputs.gz = gradZLinX[c].results.gz;
    shader[c].inputs.weightsXYZ = gradX[c].results.weightsXYZ;
    shader[c].inputs.perChipControlFlags
        = &(gradX[c].results.perChipControlFlags);
    shader[c].inputs.perPipelineControlFlags
        = gradX[c].results.perPipelineControlFlags;
shader[c].LocalSetup(c);
}

*/
for (c=0; c<numOfChips; ++c) {
    shader[c].inputs.voxel = triLinX[c].results.voxel;
    shader[c].inputs.gx = gradX[c].results.gx;
    shader[c].inputs gy = gradX[c].results.gy;
    shader[c].inputs.gz = gradZLinX[c].results.gz;
    shader[c].inputs.weightsXYZ = triLinX[c].results.weightsXYZ;
    shader[c].inputs.perChipControlFlags
        = &(triLinX[c].results.perChipControlFlags);
    shader[c].inputs.perPipelineControlFlags
        = triLinX[c].results.perPipelineControlFlags;
shader[c].LocalSetup(c);
}

for (c=0; c<numOfChips; ++c) {

```

55

```

101
5     dataSync[c].inputs.shadel = shader[c].results.shadel;
dataSync[c].inputs.weightsXYZ = shader[c].results.weightsXYZ;
dataSync[c].inputs.perPipelineControlFlags
    = shader[c].results.perPipelineControlFlags;
dataSync[c].inputs.perChipControlFlags
    = &(shader[c].results.perChipControlFlags);
dataSync[c].LocalSetup(c);
}

10  /*
15   for (c=0; c<numOfChips; ++c) {
compos[c].inputs.shadel = dataSync[c].results.shadel;
compos[c].inputs.coxel = composLinY[c].results.coxel;
compos[c].inputs.weightsXYZ = composLinY[c].results.weightsXYZ;
compos[c].inputs.perPipelineControlFlags
    = composLinY[c].results.perPipelineControlFlags;
compos[c].inputs.perChipControlFlags
    = &(composLinY[c].results.perChipControlFlags);
compos[c].LocalSetup(c);
}

20  */
25   for (c=0; c<numOfChips; ++c) {
compos[c].inputs.shadel = dataSync[c].results.shadel;
compos[c].inputs.coxel = composLinY[c].results.coxel;
compos[c].inputs.weightsXYZ = dataSync[c].results.weightsXYZ;
compos[c].inputs.perPipelineControlFlags
    = dataSync[c].results.perPipelineControlFlags;
compos[c].inputs.perChipControlFlags
    = &(dataSync[c].results.perChipControlFlags);
compos[c].LocalSetup(c);
}

30   for (c=0; c<numOfChips; ++c) {
composBuff[c].inputs.coxel = compos[c].results.coxel;
composBuff[c].inputs.perChipControlFlags
    = &(compos[c].results.perChipControlFlags);
composBuff[c].LocalSetup(c);
}

35   for (c=0; c<numOfChips; ++c) {
composSelX[c].inputs.coxel = composBuff[c].results.coxel;
composSelX[c].inputs.weightsXYZ = shader[c].results.weightsXYZ;
composSelX[c].inputs.perPipelineControlFlags
    = shader[c].results.perPipelineControlFlags;
composSelX[c].inputs.perChipControlFlags
    = &(shader[c].results.perChipControlFlags);
composSelX[c].LocalSetup(c);
}

40   for (c=0; c<numOfChips; ++c) {
composLinX[c].inputs.coxel = composSelX[c].results.coxel;
composLinX[c].inputs.weightsXYZ = composSelX[c].results.weightsXYZ;
composLinX[c].inputs.perPipelineControlFlags

```

```

102
=
composSelX[c].results.perPipelineControlFlags;
5      composLinX[c].inputs.perChipControlFlags
         = &(composSelX[c].results.perChipControlFlags);
      composLinX[c].LocalSetup(c);
}

for (c=0; c<numOfChips; ++c) {
10    composSelY[c].inputs.coxel = composLinX[c].results.coxel;
    composSelY[c].inputs.weightsXYZ = composLinX[c].results.weightsXYZ;
    composSelY[c].inputs.perPipelineControlFlags
        = composLinX[c].results.perPipelineControlFlags;
    composSelY[c].inputs.perChipControlFlags
        = &(composLinX[c].results.perChipControlFlags);
15    composSelY[c].LocalSetup(c);
}

for (c=0; c<numOfChips; ++c) {
20    composLinY[c].inputs.coxel = composLinX[c].results.coxel;
    composLinY[c].inputs.weightsXYZ = composLinX[c].results.weightsXYZ;
    composLinY[c].inputs.perPipelineControlFlags
        = composLinX[c].results.perPipelineControlFlags;
    composLinY[c].inputs.perChipControlFlags
        = &(composLinX[c].results.perChipControlFlags);
    composLinY[c].LocalSetup(c);
25}

for (c=0; c<numOfChips; ++c) {
30    finalCoxelBuff[c].inputs.coxel = compos[c].results.coxel;
    finalCoxelBuff[c].inputs.perPipelineControlFlags
        = compos[c].results.perPipelineControlFlags;
    finalCoxelBuff[c].inputs.perChipControlFlags
        = &(compos[c].results.perChipControlFlags);
    finalCoxelBuff[c].LocalSetup(&(coxMem[c]));
}

// setup geometry stuff
35 Mpd[ViewFront](1,0,0,0,
                  0,1,0,0,
                  0,0,1,0,
                  0,0,0,1);

Mdp[ViewFront](1,0,0,0,
40                  0,1,0,0,
                  0,0,1,0,
                  0,0,0,1);

Mpd[ViewLeft ](0,0,1,0,
45                  1,0,0,0,
                  0,1,0,0,
                  0,0,0,1);

Mdp[ViewLeft ](0,1,0,0,
50

```

103

```

      0,0,1,0,
      1,0,0,0,
      0,0,0,1);

```

5

```

Mpd[ ViewTop ](0,1,0,0,
      0,0,1,0,
      1,0,0,0,
      0,0,0,1);

```

10

```

Mdp[ ViewTop ](0,0,1,0,
      1,0,0,0,
      0,1,0,0,
      0,0,0,1);

```

15

```
} // constructor
```

```
Cube4::~Cube4()
```

```
{
```

20

```
    if (dataset) { delete dataset; dataset =0; }
```

```
    if (voxMem) { delete voxMem; voxMem =0; }
```

```
    if (coxMem) { delete coxMem; coxMem =0; }
```

```
    if (control) { delete control; control =0; }
```

```
    if (addrGen) { delete addrGen; addrGen =0; }
```

```
    if (memCtrl) { delete memCtrl; memCtrl =0; }
```

25

```
    if (sliceVoxelFiFo0){ delete sliceVoxelFiFo0;sliceVoxelFiFo0=0; }
```

```
    if (sliceVoxelFiFo1){ delete sliceVoxelFiFo1;sliceVoxelFiFo1=0; }
```

```
    if (triLinZ) { delete triLinZ; triLinZ =0; }
```

```
    if (triLinY) { delete triLinY; triLinY =0; }
```

```
    if (triLinX) { delete triLinX; triLinX =0; }
```

30

```
    if (gradY) { delete gradY; gradY =0; }
```

```
    if (gradX) { delete gradX; gradX =0; }
```

```
    if (gradZ) { delete gradZ; gradZ =0; }
```

```
    if (gradZLinZ) { delete gradZLinZ; gradZLinZ =0; }
```

```
    if (gradZLinY) { delete gradZLinY; gradZLinY =0; }
```

35

```
    if (gradZLinX) { delete gradZLinX; gradZLinX =0; }
```

```
    if (shader) { delete shader; shader =0; }
```

```
    if (dataSync) { delete dataSync; dataSync =0; }
```

```
    if (composBuff) { delete composBuff; composBuff =0; }
```

```
    if (compos) { delete compos; compos =0; }
```

40

```
    if (composSelX) { delete composSelX; composSelX =0; }
```

```
    if (composSelY) { delete composSelY; composSelY =0; }
```

```
    if (composLinX) { delete composLinX; composLinX =0; }
```

```
    if (composLinY) { delete composLinY; composLinY =0; }
```

```
    if (finalCoxelBuff) { delete finalCoxelBuff; finalCoxelBuff =0; }
```

```
} // destructor
```

45

```
//////////////////////////////  
// local computation functions
```

50

```
bool Cube4::RenderAnimation()
{
```

55

```

104
char str[500];

5   if (numOfFramesInAnimation == 0) {
      RenderImage();
      // MakePSfile("");
      // sprintf(str,"display img/%s.miff \n", imageRootFileName);
      // system(str);
      return true;
10 }
      PerFrameSetup();
int km=numOfFramesInAnimation,
     x=datasetSizeUVW.X(), y=datasetSizeUVW.Y(), z=datasetSizeUVW.Z();
int imgX = km * 3*x * 2, imgY = 2* 3*y;
15 if (km>8) { imgX = 8* 3*x * 2; imgY = 2* 3*y * int((km+7)/8); }

singleImagePerFrameMode = true;
viewPointSet = true;
system("/usr/bin/rm -f img/ani*.miff & \n");
sprintf(str,"convert -geometry %ix%i! img/white.miff img/dummy.miff \n",
20           imgX, imgY);
system(str);
system("display -delay 1 img/dummy.miff & \n");
double twoPi=2.0*M_PI, t=twoPi;
for (int k=0; k < km; k++) {
25   t = k/double(km-1);
   //viewPointUVW(x/2.0 + 50*cos(t*twoPi), y/2.0 + 50*sin(t*twoPi),
   z/2.0+75); // circle
   //viewPointUVW(x*t, y/2.0 , z*1.5); // leftRight
   //viewPointUVW(x/2.0 , y*t, z*1.5); // upDown
   viewPointUVW(x/2.0 , y/2.0, z/2.0 + 1/(t/z)); // near / far
30   //viewPointUVW(x/2.0 - 100*sin(t*twoPi/8.5),
   //                                y/2.0 - 100*sin(t*twoPi/8.5),
   //                                z/2.0 + 100*cos(t*twoPi/8.5));
   // 35deg circle
   cout << x<< " <<y<< " <<z<< " <<"new view point: "<< viewPointUVW <<
endl;
35   RenderImage();
   sprintf(str,"convert img/baseplaneImage.miff img/ani_%02i.miff",k);
   system(str);
   sprintf(str,"convert -draw 'image %i.%i img/baseplaneImage.miff' %s
40   %s",
           (k%8)*2*3*x, (k/8)*2*3*y, "img/dummy.miff",
           "img/dummy.miff");
           system(str);
}
   sprintf(str,"animate img/ani_*.miff & \n",k);
   system(str);
45   sprintf(str,"cp img/dummy.miff img/baseplaneImage.miff \n");
   system(str);
   sprintf(str,"diagonal view point sweep x=0...%i y=0...%i in %i
steps",x,y,km);
   MakePSfile(str);

```

50

55

```

105
    sprintf(str,"cp img/%s* ani/ \n", imageRootFileName);
    system(str);
    sprintf(str,"cp img/ani_*.miff ani/ \n");
    system(str);
    return true;
} // RenderAnimation

10 void Cube4::RenderImage()
{
    Timer timer;
    PerFrameSetup();
    RunPipeline();
    cout << "last projection " << timer << endl;
15 } // RenderImage

void Cube4::PerFrameSetup()
{
20     int c,k;

    backgroundCoxel.r = 1;
    backgroundCoxel.g = 1;
    backgroundCoxel.b = 1;
    backgroundCoxel.a = 0;
25

    backgroundShadel.r = 1;
    backgroundShadel.g = 1;
    backgroundShadel.b = 1;
    backgroundShadel.a = 0;

30     // make sure that the right dataset is in on board DRAM
    if (!datasetLoaded) {
        dataset = new LinearDataset(datasetFileName);

        // global control information
35        cout << endl << "dataset.sizeUVW = "<<dataset->sizeUVW<<endl;
        Vector3D<int> one(1,1,1);
        // make each direction a multiple of (numOfChips*blockSize)
        datasetSizeUVW = (one + (dataset->sizeUVW-
        one)/(numOfChips*blockSize))
            * (numOfChips*blockSize);
        cout << "datasetSizeUVW = "<<datasetSizeUVW
            << " number of chips = "<<numOfChips
            << " pipelines per chip = "<<numOfPipelinesPerChip
            << " block size = "<<blockSize
            << endl;
40

45     // initialize memory modules
        for (c=0; c<numOfChips; ++c) {
            voxMem[c].Init((datasetSizeUVW.U() *
50             datasetSizeUVW.V() *

```

106

```

datasetSizeUVW.W() / numOfChips);
}

5

datasetToPipelineMatrix = Mp[ViewFront];
pipelineToDatasetMatrix = Mdp[ViewFront];

10 // initialize address generator for writing data
for (c=0; c<numOfChips; ++c) {
    addrGen[c].PerFrameSetup(datasetSizeUVW,
pipelineToDatasetMatrix);
}

15 // move dataset into memory modules
int index,address;
int v_step(dataset->sizeUVW.U());
int w_step(v_step * dataset->sizeUVW.V());
for (int w=0; w<datasetSizeUVW.W(); ++w) {
    for (int v=0; v<datasetSizeUVW.V(); ++v) {
        for (int u=0; u<datasetSizeUVW.U(); ++u) {
            index = addrGen[0].ChipIndex(u,v,w);
            address = addrGen[0].MemoryAddress(u,v,w);
            if (u < dataset->sizeUVW.U() &&
                v < dataset->sizeUVW.V() &&
                w < dataset->sizeUVW.W())
                voxMem[index](address) = dataset-
20 >voxelData[u + v*v_step + w*w_step];
            else
                voxMem[index](address) = 0;
//voxMem[index](address) = u + v*32 + w*32*32;
//voxMem[index](address) = index*256/numOfChips;
//if (v==w) voxMem[index](address) = 200;
        }
    }
}
datasetLoaded = true;

25 // free temporary storage
delete dataset; dataset=0;
}

30 /////////////////
// get the viewdependent global variables

35 if (!viewPointSet) {
    if (compositingStyle == BackToFront) {
        viewPointUVW(datasetSizeUVW.X()/2.0,
40 datasetSizeUVW.Y()/2.0,
                    2.0*datasetSizeUVW.Z());
    }
    else if (compositingStyle == FrontToBack) {
        viewPointUVW(datasetSizeUVW.X()/2.0,
                    datasetSizeUVW.Y(),
                    datasetSizeUVW.Z());
    }
}

45

```

50

55

```

datasetSizeUVW.Y() / 2.0,
5                               -2.0 * datasetSizeUVW.Z());
}
}

Vector3D<FixPointNumber> sightRay(datasetSizeUVW);
sightRay /= 2.0;
10 cout << "datasetCenter:" << sightRay << " viewPointUVW:" << viewPointUVW;
sightRay -= viewPointUVW;
Vector3D<FixPointNumber> absSightRay(sightRay.U().Abs(),
15                               sightRay.V().Abs(),
sightRay.W().Abs());

int maxDirection, maxIndex(0);
if (absSightRay[maxIndex] < absSightRay[1]) maxIndex = 1;
20 if (absSightRay[maxIndex] < absSightRay[2]) maxIndex = 2;

switch (maxIndex) {
case 0: maxDirection = ViewLeft ; break;
case 1: maxDirection = ViewTop ; break;
25 case 2: maxDirection = ViewFront; break;
}

datasetToPipelineMatrix = Mdp[maxDirection];
PipelineToDatasetMatrix = Mpd[maxDirection];

30 datasetSizeXYZ = datasetToPipelineMatrix * datasetSizeUVW;
viewPointXYZ    = datasetToPipelineMatrix * viewPointUVW;

if (viewPointXYZ.Z() > datasetSizeXYZ.Z() / 2.0) {
    // view point behind dataset
    compositingStyle = BackToFront;
35 }
else {
    // view point in front of dataset
    compositingStyle = FrontToBack;
}

40 cout << " sightRay:" << sightRay
        << " viewPointXYZ:" << viewPointXYZ
        << " Processing:" << TypeStr::MajorViewDirection[maxDirection]
        << " Compos:" << TypeStr::CompositingStyle[compositingStyle]
        << endl;

45 // transform light sources
lightSourceXYZ = lightSourceUVW;
for (k=0; k<lightSourceXYZ.Size(); ++k) {
    lightSourceXYZ[k].TransformDirection(datasetToPipelineMatrix);
}
50

```

```

// initialize address generator for reading data
5      for (c=0; c<numOfChips; ++c) {
            addrGen[c].PerFrameSetup(datasetSizeUVW, pipelineToDatasetMatrix);
        }

        // initialize on chip control
        // last voxel of volume=>wrap around to first
        // (reverse control IntVoxelPos computation)
10     Vector3D<int> startVoxelPos(-(1+blockSize)*(blockSize*numOfChips), -1, -1);

        int nextX;
        for (c=0; c<numOfChips; ++c) {
            control[c].PerFrameSetup(datasetSizeXYZ, startVoxelPos,
15           viewPointXYZ, compositingStyle,
            projectionStyle, cubeMode);
            nextX = startVoxelPos.X() + blockSize;
            startVoxelPos.SetX(nextX);
        }

#define ForAllChips for (c=0; c<numOfChips; ++c)

20     ForAllChips memCtrl[c].PerFrameSetup();
25     ForAllChips sliceVoxelFiFo0[c].PerFrameSetup();
     ForAllChips sliceVoxelFiFo1[c].PerFrameSetup();
     ForAllChips triLinZ[c].PerFrameSetup();
     ForAllChips triLinY[c].PerFrameSetup();
     ForAllChips triLinX[c].PerFrameSetup();
     ForAllChips gradY[c].PerFrameSetup();
30     ForAllChips gradX[c].PerFrameSetup();
     ForAllChips gradZ[c].PerFrameSetup();
     ForAllChips gradZLinZ[c].PerFrameSetup();
     ForAllChips gradZLinY[c].PerFrameSetup();
     ForAllChips gradZLinX[c].PerFrameSetup();
35     ForAllChips shader[c].PerFrameSetup();
     ForAllChips dataSync[c].PerFrameSetup();
     ForAllChips composBuff[c].PerFrameSetup();
     ForAllChips compos[c].PerFrameSetup();
     ForAllChips composSelX[c].PerFrameSetup();
     ForAllChips composSelY[c].PerFrameSetup();
40     ForAllChips composLinX[c].PerFrameSetup();
     ForAllChips composLinY[c].PerFrameSetup();
     ForAllChips finalCoxelBuff[c].PerFrameSetup();

        DebugPerFrameSetup();
45 } // PerFrameSetup

void Cube4::RunPipeline()
{
    cout << endl << "running pipeline ... " << endl;
50

```

109

```

int c;

5      #define ForAllChips    for (c=0; c<numOfChips; ++c)

/*
   // precomputed clk delay
   // now determined on the fly in FinalCoxelBuff::RunForOneClockCycle()
10     int clk_end;
   // volume + 2 Slices delay
   clk_end =
datasetSizeXYZ.X()*datasetSizeXYZ.Y()*(datasetSizeXYZ.Z()+2+blockSize);
   clk_end /= (numOfChips*numOfPipelinesPerChip);

15     // 17 pipeline stages and 4 beam buffer delay
   clk_end += 17+4;

   // final pixel dribbel delay
   clk_end += 4*blockSize;
20 */
for (clk=0; clk>=0; ++clk) {

   //ForAllChips finalCoxelBuff[c].RunForOneClockCycle();
   ForAllChips compos[c].RunForOneClockCycle();
25     ForAllChips composLinY[c].RunForOneClockCycle();
   ForAllChips composSelY[c].RunForOneClockCycle();
   ForAllChips composLinX[c].CommunicateForOneClockCycle();
   ForAllChips composLinX[c].RunForOneClockCycle();
   ForAllChips composSelX[c].CommunicateForOneClockCycle();
30     ForAllChips composSelX[c].RunForOneClockCycle();
   ForAllChips composBuff[c].RunForOneClockCycle();
   ForAllChips compos[c].WriteResultsToPipelineRegister();
   ForAllChips dataSync[c].RunForOneClockCycle();
   ForAllChips shader[c].RunForOneClockCycle();
35     //ForAllChips gradZLinX[c].RunForOneClockCycle();
   //ForAllChips gradZLinY[c].RunForOneClockCycle();
   //ForAllChips gradZLinZ[c].RunForOneClockCycle();
   //ForAllChips gradZ[c].RunForOneClockCycle();
   //ForAllChips gradX[c].RunForOneClockCycle();
   //ForAllChips gradY[c].RunForOneClockCycle();
40     ForAllChips triLinX[c].CommunicateForOneClockCycle();
   ForAllChips triLinX[c].RunForOneClockCycle();
   ForAllChips triLinY[c].RunForOneClockCycle();
   ForAllChips triLinZ[c].RunForOneClockCycle();
   ForAllChips sliceVoxelFiFo1[c].RunForOneClockCycle();
   ForAllChips sliceVoxelFiFo0[c].RunForOneClockCycle();
45     ForAllChips memCtrl[c].RunForOneClockCycle();
   ForAllChips addrGen[c].RunForOneClockCycle();
   ForAllChips control[c].RunForOneClockCycle();

   //DebugControl(clk);
   //DebugAddrGen(clk);
50     //DebugMemCtrl(clk);

```

55

```

110          //DebugSliceVoxelFiFo0(clk);
5           //DebugSliceVoxelFiFo1(clk);
           //DebugTriLinZ(clk);
           //DebugTriLinY(clk);
           //DebugTriLinX(clk);
           //DebugGradY(clk);
           //DebugGradX(clk);
10          //DebugGradZ(clk);
           //DebugGradZLinZ(clk);
           //DebugGradZLinY(clk);
           //DebugGradZLinX(clk);
           //DebugShader(clk);
           //DebugDataSync(clk);
15          //DebugComposSelX(clk);
           //DebugComposLinX(clk);
           //DebugComposSelY(clk);
           //DebugComposLinY(clk);
           //DebugCompos(clk);
           DebugComposBuffer(clk);
20          //DebugFinalCoxelBuffer(clk);
      } // clk loop

      // GetFinalImage();
} // RunPipeline
25
// end of Cube4.C
::::::::::
cube4/Cube4.h
::::::::::
// Cube4.h
30 // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

35 // Cube4 complete parallel rendering pipeline
// in : dataset, colorTable, illumTable
// out: baseplane image

40 #ifndef _Cube4_h_ // prevent multiple includes
#define _Cube4_h_

45 #include <fstream.h>    // strcmp
#include <iostream.h>    // strcmp
#include <string.h>       // strcmp
#include <strstream.h> // strstream
#include <stdlib.h>        // atoi
#include <stdio.h>         // sprintf
#include <assert.h>
#include "Object.h"
#include "Misc.h"
50 #include "DynaArray.h"
#include "FixPointNumber.h"

```

```

5      #include "Vector3D.h"
      #include "Matrix4x4.h"
      #include "FiFo.h"
      #include "Timer.h"
      #include "Voxel.h"
      #include "Shadel.h"
      #include "Coxel.h"
      #include "Light.h"
      #include "LinearDataset.h"
      #include "VoxMem.h"
      #include "CoxMem.h"
      #include "Control.h"
      #include "AddressGenerator.h"
      #include "MemoryCtrl.h"
      #include "SliceVoxelFiFoStage.h"
      #include "SliceVoxelFiFoPipeline.h"
      #include "TriLinZStage.h"
      #include "TriLinZPipeline.h"
      #include "TriLinYStage.h"
      #include "TriLinYPipeline.h"
      #include "TriLinXStage.h"
      #include "TriLinXPipeline.h"
      #include "GradyStage.h"
      #include "GradyPipeline.h"
      #include "GradXStage.h"
      #include "GradXPipeline.h"
      #include "GradZStage.h"
      #include "GradZPipeline.h"
      #include "GradZLinZStage.h"
      #include "GradZLinZPipeline.h"
      #include "GradZLinYStage.h"
      #include "GradZLinYPipeline.h"
      #include "GradZLinXStage.h"
      #include "GradZLinXPipeline.h"
      #include "ShaderStage.h"
      #include "ShaderPipeline.h"
      #include "DataSyncStage.h"
      #include "DataSyncPipeline.h"
      #include "ComposBufferStage.h"
      #include "ComposBufferPipeline.h"
      #include "ComposStage.h"
      #include "ComposPipeline.h"
      #include "ComposSelXStage.h"
      #include "ComposSelXPipeline.h"
      #include "ComposSelYStage.h"
      #include "ComposSelYPipeline.h"
      #include "ComposLinXStage.h"
      #include "ComposLinXPipeline.h"
      #include "ComposLinYStage.h"
      #include "ComposLinYPipeline.h"
      #include "FinalCoxelBuffer.h"

```

```

112
class Cube4 : virtual public Object {
public:
    static void      Demo ();
5
    // constructors & destructors
    Cube4(int argc=0, char *argv[]={});
    virtual ~Cube4();

10   // show/set data & data properties
    virtual      bool      PrintUsage()           const;
    virtual /*inline*/ bool      SetDatasetFile (const char * fileName);
    virtual /*inline*/ bool      SetColorTableFile(const char * fileName);
    virtual /*inline*/ bool      SetAlphaTableFile(const char * fileName);
    virtual /*inline*/ bool      SetReflectanceTableFile(const char *
15   fileName);
    virtual /*inline*/ bool      SetImageFileRoot (const char * fileName);
    virtual      bool      LoadColorTable();
    virtual      bool      LoadAlphaTable();
    virtual      bool      LoadReflectanceTable();

20   // local computation functions
    virtual bool RenderAnimation();
    virtual void RenderImage();

25   protected:
    // internal utility functions
    virtual void PerFrameSetup();
    virtual void RunPipeline();

    // Debugging tools
30   virtual void DebugImage(const PerChipControlFlags & flags);
    virtual void DebugWritePixel( const int c,
                                 const Vector3D<int> & voxelPosXYZ,
                                 const unsigned char red,
35   const unsigned char green,
                                 const unsigned char blue);

    // Debugging functions for each pipeline stage
    void Cube4::DebugPerFrameSetup();
    bool Cube4::ClkInInterval(const int clk,
40
        const int pipelineStart,
        const int startBeam,
        const int startSlice,
        const int timeWindowSize);
    virtual void DebugControl(const int clk);
    virtual void DebugAddrGen(const int clk);
    virtual void DebugMemCtrl(const int clk);
50

```

```

113
virtual void DebugSliceVoxelFiFo0(const int clk);
5   virtual void DebugSliceVoxelFiFo1(const int clk);
virtual void DebugTriLinZ(const int clk);
virtual void DebugTriLinY(const int clk);
virtual void DebugTriLinX(const int clk);
virtual void DebugGradY (const int clk);
virtual void DebugGradX (const int clk);
10  virtual void DebugGradZ (const int clk);
virtual void DebugGradZLinZ(cohost int clk);
virtual void DebugGradZLinY(const int clk);
virtual void DebugGradZLinX(const int clk);
virtual void DebugShader(const int clk);
virtual void DebugDataSync(const int clk);
15  virtual void DebugCompos (const int clk);
virtual void DebugComposBuffer (const int clk);
virtual void DebugComposSelX (const int clk);
virtual void DebugComposSelY (const int clk);
virtual void DebugComposLinX (const int clk);
virtual void DebugComposLinY (const int clk);
20  virtual void DebugFinalCoxelBuffer (const int clk);

        virtual bool ReadCube4PipelineParametersFromFile(const char *
srcFileName);
        virtual bool ReadCube4PipelineParametersFromCommandLine(int argc, char
*argv[]);
25    virtual bool SetParameter(const char * id, const char * str );
        virtual bool SetDatasetToImageMatrix(const char * str );
        virtual bool SetAmbientIntensity(const char * str );
        virtual bool SetViewPoint(const char * str );

30    virtual void GetFinalImage();
        virtual void MakePSfile(const char * comment);

        ProjectionStyles projectionStyle;
        CompositingStyles compositingStyle;
35    ShaderModes      shaderMode;
        CubeModes       cubeMode;

        LinearDataset *dataset;

        char datasetFileName[200],
40      colorTableFileName[200],
          alphaTableFileName[200],
          reflectanceTableFileName[200],
          imageRootFileName[200];

        bool datasetLoaded,
45      colorTableLoaded,
          alphaTableLoaded,
          reflectanceTableLoaded;

        int numOfChips, numOfPipelinesPerChip, blockSize; // c p b
50

```

```

114
    int numOfPartialBlockBeams,           numOfBlockBeams, numOfSlices; // Nb
NB Ns
5     int numOfClksPerBeam, numOfClksPerSlice, numOfClksPerVolume; // ccB ccS
ccV

    Vector3D<int> datasetSizeXYZ;
    Vector3D<int> datasetSizeUVW;

10    // animation setup
    int numOfFramesInAnimation;

    // view parameters
    Matrix4x4<int> Mpd[3], Mdp[3];
    Matrix4x4<int> pipelineToDatasetMatrix, datasetToPipelineMatrix;
15    Vector3D<FixPointNumber> viewPointUVW;
    Vector3D<FixPointNumber> viewPointXYZ;
    bool viewPointSet;

    // shading parameters
20    DynaArray<Light> lightSourceXYZ;
    DynaArray<Light> lightSourceUVW;
    Vector3D<double> IAmbient;
    double phongExponent;
    double Ka;
25    double Kd;
    double Ks;
    Coxel backgroundCoxel;
    Shadel backgroundShadel;

    // trilin parameters
30    int trilinWeightBits;

    // gradient parameters
    int gradientBits;

    // shader parameters
35    int shadelColorBits;
    int shadelAlphaBits;
    int mapWeightBits;
    bool reflectanceMapIP;
    bool distortionCompensation;
    bool constantAngularResolution;
40    int reflectanceMapBits;
    int atanTableBits;
    int colorTableBits;
    int alphaTableBits;

45    // Alpha Lookup Function
    double threshold;
    double maxWidth;
    double mulAlpha;

50    // compos parameters

```

115

```

5      int coxelColorBits;
      int coxelAlphaBits;
      int composWeightBits;

10     // external DRAM memory
      VoxMem           *voxMem;
      CoxMem           *coxMem;

15     // Cube4 pipeline stages
      Control          *control;
      AddressGenerator *addrGen;
      MemoryCtrl       *memCtrl;
      SliceVoxelFiFoStage *sliceVoxelFiFo0;
      SliceVoxelFiFoStage *sliceVoxelFiFo1;
      TriLinZStage     *triLinZ;
      TriLinYStage     *triLinY;
      TriLinXStage     *triLinX;
      GradYStage       *gradY;
      GradXStage       *gradX;
      GradZStage       *gradZ;
      GradZLinZStage   *gradZLinZ;
      GradZLinYStage   *gradZLinY;
      GradZLinXStage   *gradZLinX;
      ShaderStage       *shader;
      DataSyncStage    *dataSync;
      ComposBufferStage *composBuff;
      ComposStage       *compos;
      ComposSelXStage   *composSelX;
      ComposSelYStage   *composSelY;
      ComposLinXStage   *composLinX;
      ComposLinYStage   *composLinY;
      FinalCoxelBuffer *finalCoxelBuff;

      int clk;

35     // class variables for debugging tools
      unsigned char *volumeBuffer;
      int sliceSize, volumeSize, slabSize, x_step, y_step, z_step;
      Vector3D<int> offset;
      int partialBeam, beam, blockSlice, blockBeam, blockSlab, startClk, endClk;
      bool singleImagePerFrameMode;

40     friend class Control;
     friend class SliceVoxelFiFoPipeline;
     friend class TriLinZStage;
     friend class TriLinZPipeline;
     friend class TriLinYStage;
     friend class TriLinYPipeline;
     friend class TriLinXStage;
     friend class TriLinXPipeline;
     friend class GradZLinZPipeline;
     friend class GradZLinYPipeline;
     friend class GradZLinXPipeline;

```

55

```

5           friend class GradYPipeline;
friend class GradXPipeline;
friend class ShaderPipeline;
friend class ShaderStage;
friend class DataSyncStage;
friend class DataSyncPipeline;
friend class ComposStage;
friend class ComposPipeline;
10          friend class ComposBufferStage;
friend class ComposBufferPipeline;
friend class ComposSelXStage;
friend class ComposSelXPipeline;
friend class ComposSelYStage;
15          friend class ComposSelYPipeline;
friend class ComposLinXStage;
friend class ComposLinXPipeline;
friend class ComposLinYStage;
friend class ComposLinYPipeline;
20          friend class FinalCoxelBuffer;
}; // Cube4

#ifndef      // _Cube4_h_
:::::::::::
25          cube4/Cube4Debug.C
:::::::::::
// Cube4Debug.C
// (c) Ingmar Bitter '97

30          // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

// to be included from Cube4.C
// holds all the (unnecessary) :-) debug functions

35          #define RPV results.perPipelineControlFlags->valid

void Cube4::DebugPerFrameSetup()
{
40          partialBeam = blockSize/numOfPipelinesPerChip;
beam = partialBeam * datasetSizeXYZ.X()/blockSize/numOfChips;
blockSlice = beam * blockSize;
blockBeam = blockSlice * blockSize;
blockSlab = blockBeam * datasetSizeXYZ.Y()/blockSize;
} // DebugPerFrameSetup

50          bool Cube4::ClkInInterval(const int clk,
                           const int pipelineStart,
                           const int startBeam,
                           const int startSlice,
                           const int timeWindowSize)
{

```

```

117
startClk = pipelineStart +
    startBeam /blockSize*blockBeam + startBeam %blockSize*beam +
    startSlice/blockSize*blockSlab + startSlice%blockSize*blockSlice;
5      endClk = startClk + timeWindowSize;

    return (startClk <= clk && clk <= endClk);
} // ClkInInterval

10 #define ForAll_c_p( bla ) { for (c=0; c<numOfChips; ++c) { for (p=0;
p<numOfPipelinesPerChip; ++p) { bla ; } } }
#define COUT_c_p(w, bla) for (c=0; c<numOfChips; ++c) { for (p=0;
p<numOfPipelinesPerChip; ++p) { cout.width(w); bla } cout<<" ";}
#define COUT_p(chip, w, bla)           { c=(chip); for (p=0;
p<numOfPipelinesPerChip; ++p) { cout.width(w); bla } cout<<" ";}
15

void Cube4::DebugControl(const int clk)
{
    int c,p;
    /*
20        DebugImage(control[0].results.perChipControlFlags);
        ForAll_c_p( DebugWritePixel
        (c, control[c].results.perPipelineControlFlags[p].voxelPosXYZ,
         0,
         (unsigned char) 128.0 *
25       (1.0 + control[c].results.weightsXYZ[p].X()),
         (unsigned char) 128.0 *
       (1.0 + control[c].results.weightsXYZ[p].Y())));
        */
        if (ClkInInterval(/* current clock */ clk,
                           /* pipeline start */ 0,
                           /* beam/slice start */ 0, 0,
                           /* time window size */ 2*beam)
            ||
            ClkInInterval(clk, 0, blockSize-1,0,2*beam)
            ||
            ClkInInterval(clk, 0, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
            )
30        {
            cout<<endl; cout.width(4); cout << clk << " Ctrl:";
            cout<<" left="; for (c=0; c<1; ++c) cout<<control[c].leftmostChipIndex<<" ";
            cout<<" vox";
            cout<<" X=";   COUT_c_p( 3, cout<<control[c].results.voxelPosXYZ[p].X(); );
            cout<<" Y=";   COUT_c_p( 2, cout<<control[c].results.voxelPosXYZ[p].Y(); );
            cout<<" Z=";   COUT_c_p( 2, cout<<control[c].results.voxelPosXYZ[p].Z(); );
40            //cout<<" clk=";COUT_c_p( 3,
            cout<<control[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
            //cout << "\tweights[0] = " << control[0].results.weightsXYZ[0];
            //cout << "\tweights[1] = " << control[0].results.weightsXYZ[1];
            //cout << "\tendOfRay[0] = " <<
45            control[0].results.perPipelineControlFlags[0].endOfRay;
            //cout << "\tendOfRay[1] = " <<
            control[0].results.perPipelineControlFlags[1].endOfRay;
        }
50

```

55

```

    118
} // DebugControl

5   void Cube4::DebugAddrGen(const int clk)
{
    int c,p;

    DebugImage(addrGen[0].results.perChipControlFlags);
    ForAll_c_p( DebugWritePixel
                (c,
     addrGen[c].results.perPipelineControlFlags[p].voxelPosXYZ,
                                addrGen[c].results.address[p], 0, 0)
    );

15  if (addrGen[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "addrGen::volumeStart@inputs@" << clk << " ";
    if (addrGen[0].results.perChipControlFlags.volumeStart)
    cout << endl << "addrGen::volumeStart@results@" << clk << " ";

    int volStart=1;
20  if (ClkInInterval(/* current clock      */ clk,
                    /* pipeline start      */ volStart,
                    /* beam/slice start */ 0, 0,
                    /* time window size */ 2*beam)
        || ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
        || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
    ) {
        cout<<endl; cout.width(4); cout << clk << " AddrGen:";
        cout<<" vox";
        cout<<" X="; COUT_c_p( 3,
cout<<addrGen[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
        cout<<" Y="; COUT_c_p( 2,
cout<<addrGen[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
        cout<<" Z="; COUT_c_p( 2,
cout<<addrGen[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
        cout<<" addr[0:1]=";
        <<addrGen[0].results.address[1]
        <<"\t"<<addrGen[0].results.perPipelineControlFlags[1].voxelPosXYZ;
    }
} // DebugAddrGen

40  void Cube4::DebugMemCtrl(const int clk)
{
    int c,p;

    DebugImage(memCtrl[0].results.perChipControlFlags);
    ForAll_c_p( DebugWritePixel
                (c,
     memCtrl[c].results.perPipelineControlFlags[p].voxelPosXYZ,
                                memCtrl[c].results.voxel[p].raw16bit,
0, 0) );
}

```

50

55

```

119
    if (memCtrl[0].inputs.perChipControlFlags-
>volumeStart)
        cout << endl << "memCtrl::volumeStart@inputs@" << clk << " ";
    if (memCtrl[0].results.perChipControlFlags.volumeStart)
        cout << endl << "memCtrl::volumeStart@results@" << clk << " ";

    int volStart=2;
    if (ClkInInterval(/* current clock      */ clk,
10          /* pipeline start     */ volStart,
          /* beam/slice start */ 0, 0,
          /* time window size */ 2*beam)
        || ClkInInterval(clk, volStart, blockSize-2,0,3*beam)
        //|| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
        )
    {
        cout<<endl; cout.width(4); cout << clk << " MemCtrl:";
        cout<<" ctrl";
        cout<<" X="; COUT_c_p( 3,
cout<<memCtrl[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
        cout<<" Y="; COUT_c_p( 2,
cout<<memCtrl[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
20        cout<<" Z="; COUT_c_p( 2,
cout<<memCtrl[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
        cout<<" v[0:1]="
            <<memCtrl[0].results.voxel[1]
            <<memCtrl[0].results.perPipelineControlFlags[1].voxelPosXYZ;
        /*
        cout<<"\n\t\tvvox";
        cout<<" X="; COUT_c_p( 3, cout<<memCtrl[c].results.voxel[p].raw16bit%32; );
        cout<<" Y="; COUT_c_p( 2, cout<<memCtrl[c].results.voxel[p].raw16bit/32%32;
    );
        cout<<" Z="; COUT_c_p( 2,
30        cout<<memCtrl[c].results.voxel[p].raw16bit/32/32%32; );
        cout<<" v[0:1]="
            <<memCtrl[0].results.voxel[1]
            <<memCtrl[0].results.perPipelineControlFlags[1].voxelPosXYZ;
        */
        //cout<<"\n\t\tvvox";
        //cout<<" V=";      COUT_c_p( 3, cout<<memCtrl[c].results.voxel[p].raw16bit;
    );
    /*
        cout << "in: ";
        cout << "\tout: ";
        cout << " voxel[0] = " << memCtrl[0].results.voxel[0]
        << memCtrl[0].results.perPipelineControlFlags[0].voxelPosXYZ;
        << memCtrl[0].results.perPipelineControlFlags[1].voxelPosXYZ;
        */
    }
} // DebugMemCtrl
45
void Cube4::DebugSliceVoxelFiFo0(const int clk)
{
    int c,p;
50

```

55

120

```

DebugImage(memCtrl[0].results.perChipControlFlags);
ForAll_c_p( DebugWritePixel
5           (c,
sliceVoxelFiFo0[c].results.perPipelineControlFlags[p].voxelPosXYZ,
(sliceVoxelFiFo0[c].results.voxel[p].raw16bit),0 ,0); );

if (sliceVoxelFiFo0[0].inputs.perChipControlFlags->volumeStart)
cout << endl << "sliceVoxelFiFo0::volumeStart@inputs@" << clk << " ";
if (sliceVoxelFiFo0[0].results.perChipControlFlags.volumeStart)
cout << endl << "sliceVoxelFiFo0::volumeStart@results@" << clk << " ";

int volStart=3;
15 if (ClkInInterval(/* current clock      */ clk,
                  /* pipeline start     */ volStart,
                  /* beam/slice start */ 0, 0,
                  /* time window size */ 2*beam)
    || ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
    || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
20   ) {
    cout<<endl; cout.width(4); cout << clk << " SliceFiFo0:";
    cout<<" ctrl";
    cout<<" X="; COUT_c_p( 3,
cout<<sliceVoxelFiFo0[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
25   cout<<" Y="; COUT_c_p( 2,
cout<<sliceVoxelFiFo0[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
    cout<<" Z="; COUT_c_p( 2,
cout<<sliceVoxelFiFo0[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
    cout<<" v[0:1]="
      <<sliceVoxelFiFo0[0].results.voxel[1]
      <<sliceVoxelFiFo0[0].results.perPipelineControlFlags[1].voxelPosXYZ;
30   }
} // DebugSliceVoxelFiFo0

void Cube4::DebugSliceVoxelFiFo1(const int clk)
35 {
    int c,p;

    DebugImage(memCtrl[0].results.perChipControlFlags);
    ForAll_c_p( DebugWritePixel
40           (c,
sliceVoxelFiFo1[c].results.perPipelineControlFlags[p].voxelPosXYZ,
(sliceVoxelFiFo1[c].results.voxel[p].raw16bit),0 ,0); );

    if (sliceVoxelFiFo1[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "sliceVoxelFiFo1::volumeStart@inputs@" << clk << " ";
45    if (sliceVoxelFiFo1[0].results.perChipControlFlags.volumeStart)
        cout << endl << "sliceVoxelFiFo1::volumeStart@results@" << clk << " ";

    int volStart=4+blockSlice;

```

50

55

```

121
if (ClkInInterval(/* current clock */ clk,
                  /* pipeline start */ volStart,
                  /* beam/slice start */ 0, 0,
                  /* time window size */ 2*beam)
    || ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
    || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
    ) {
    cout<<endl; cout.width(4); cout << clk << " SliceFiFol:";
    cout<<" ctrl";
    cout<<" X="; COUT_c_p( 3,
cout<<sliceVoxelFiFol[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
    cout<<" Y="; COUT_c_p( 2,
cout<<sliceVoxelFiFol[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
    cout<<" Z="; COUT_c_p( 2,
15   cout<<sliceVoxelFiFol[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
    cout<<" v[0:1]="
        <<sliceVoxelFiFol[0].results.voxel[1]
        <<sliceVoxelFiFol[0].results.perPipelineControlFlags[1].voxelPosXYZ;
    }
} // DebugSliceVoxelFiFol
20

void Cube4::DebugTriLinZ(const int clk)
{
    int c,p;

25    DebugImage(triLinZ[0].results.perChipControlFlags);
    ForAll_c_p(DebugWritePixel
                (c,
triLinZ[c].results.perPipelineControlFlags[p].voxelPosXYZ,
                (triLinZ[c].triLinZPipeline[p].results.voxel->raw16bit),0 ,0); );

    if (triLinZ[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "triLinZ::volumeStart@inputs@" << clk << " ";
    if (triLinZ[0].results.perChipControlFlags.volumeStart)
        cout << endl << "triLinZ::volumeStart@results@" << clk << " ";

35    int volStart=5+2*blockSlice;
    if (ClkInInterval(/* current clock */ clk,
                      /* pipeline start */ volStart,
                      /* beam/slice start */ 0, 0,
                      /* time window size */ 2*beam+blockSlab+blockSlice)
        || ClkInInterval(clk, volStart, blockSize-2,0,3*beam)
        //|| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
        ) {
        cout<<endl; cout.width(4); cout << clk << " TriZ:";
40    /*
        cout<<" vox";
        cout<<" X="; COUT_c_p( 3,
cout<<triLinZ[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );

```

50

55

```

122
    cout<<" Y="; COUT_c_p( 2,
cout<<triLinZ[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
    cout<<" Z="; COUT_c_p( 2,
5      cout<<triLinZ[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
    cout<<" v[0:1]="
        <<Voxel(triLinZ[0].triLinZPipeline[1].a)
        <<Voxel(triLinZ[0].triLinZPipeline[1].b);
/*
10     if (0 == triLinZ[1].results.weightsXYZ[1].Z()) cout<<" out=vox A:";
     else
        cout<<" w=1 vox A:";
     cout<<" X="; COUT_c_p(3, cout<<Voxel(triLinZ[c].triLinZPipeline[p].a).X());
);
     cout<<" Y="; COUT_c_p(2, cout<<Voxel(triLinZ[c].triLinZPipeline[p].a).Y());
15     cout<<" Z="; COUT_c_p(2, cout<<Voxel(triLinZ[c].triLinZPipeline[p].a).Z());
);
        cout<<
Z=<<triLinZ[1].results.perPipelineControlFlags[1].voxelPosXYZ.Z() << "(Ctrl)";
if (0 == triLinZ[1].results.weightsXYZ[1].Z()) cout<<"\n\t w=0 vox
20 B:"; else
        cout<<"\n\t out=vox B:";
     cout<<" X="; COUT_c_p(3, cout<<Voxel(triLinZ[c].triLinZPipeline[p].b).X());
);
     cout<<" Y="; COUT_c_p(2, cout<<Voxel(triLinZ[c].triLinZPipeline[p].b).Y());
25     cout<<" Z="; COUT_c_p(2, cout<<Voxel(triLinZ[c].triLinZPipeline[p].b).Z());
);
}
} // DebugTriLinZ

30 void Cube4::DebugTriLinY(const int clk)
{
    int c,p;

    DebugImage(triLinY[0].results.perChipControlFlags);
    ForAll_c_p(DebugWritePixel
35             (c,
triLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ,
            (triLinY[c].triLinYPipeline[p].results.voxel->raw16bit),0 ,0););
    if (triLinY[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "triLinY::volumeStart@inputs@" << clk << " ";
    if (triLinY[0].results.perChipControlFlags.volumeStart)
        cout << endl << "triLinY::volumeStart@results@" << clk << " ";

40    int volStart=6+2*blockSlice+beam;
    if (ClkInInterval(/* current clock */ clk,
                     /* pipeline start */ volStart,
                     /* beam/slice start */ 1, 1,
                     /* time window size */ blockSize)
45
50

```

```

123
    || ClkInInterval(clk, volStart,      datasetSizeXYZ.Y()-2,blockSize-
1,2*blockSize+3*beam)
    || ClkInInterval(clk, volStart, blockSize,blockSize,blockSize+2*beam)
    ) {
5       cout<<endl; cout.width(4); cout << clk << " TriY:";
       //cout<< " vox";
       //cout<< " X=";      COUT_c_p( 3,
10      cout<<triLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
       //cout<< " Y=";      COUT_c_p( 2,
       cout<<triLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
       //cout<< " Z=";      COUT_c_p( 2,
       cout<<triLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
       //cout<< " v[0:1]="
       //    <<Voxel(triLinY[0].triLinYPipeline[1].a)
15      //    <<Voxel(triLinY[0].triLinYPipeline[1].b)
       //
       <<"Wy=<<triLinY[0].triLinYPipeline[1].results.weightsXYZ->Y();
           if (0 == triLinY[1].results.weightsXYZ[1].Z()) cout<< " out=vox A:";
           else
               cout<< " w=1 vox A:";
20      if (!triLinY[1].triLinYPipeline[1].results.perPipelineControlFlags-
>valid) cout<< "invalid"; else (
               cout<< " X="; COUT_c_p(3,
       cout<<Voxel(triLinY[c].triLinYPipeline[p].a).X(); );
               cout<< " Y="; COUT_c_p(2,
25      cout<<Voxel(triLinY[c].triLinYPipeline[p].a).Y(); );
               cout<< " Z="; COUT_c_p(2,
       cout<<Voxel(triLinY[c].triLinYPipeline[p].a).Z(); );
               }
               if (0 == triLinY[1].results.weightsXYZ[1].Z()) cout<< "\n\t w=0 vox
B:"; else
30      cout<< "\n\t out=vox B:";
               if (!triLinY[1].triLinYPipeline[1].results.perPipelineControlFlags-
>valid) cout<< "invalid"; else (
               cout<< " X="; COUT_c_p(3,
       cout<<Voxel(triLinY[c].triLinYPipeline[p].b).X(); );
               cout<< " Y="; COUT_c_p(2,
35      cout<<Voxel(triLinY[c].triLinYPipeline[p].b).Y(); );
               cout<< " Z="; COUT_c_p(2,
       cout<<Voxel(triLinY[c].triLinYPipeline[p].b).Z(); );
               cout<< "
Z=<<triLinY[1].results.perPipelineControlFlags[1].voxelPosXYZ.Z() << "(Ctrl1)";
40      }
    )
} // DebugTriLinY

void Cube4::DebugTriLinX(const int clk)
45
{
    int c,p;
    DebugImage(triLinX[0].results.perChipControlFlags);
    ForAll_c_p(DebugWritePixel

```

50

55

```

124
(c,
triLinX[c].results.perPipelineControlFlags[p].voxelPosXYZ,
5      (triLinX[c].triLinXPipeline[p].results.voxel->raw16bit),0,0);;

if (triLinX[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "triLinX::volumeStart@inputs@" << clk << " ";
if (triLinX[0].results.perChipControlFlags.volumeStart)
    cout << endl << "triLinX::volumeStart@results@" << clk << " ";

10     int volStart=6 + 2*blockSlice + beam;
if (ClkInInterval(/* current clock */ clk,
                  /* pipeline start */ volStart,
                  /* beam/slice start */ 1, 1,
15                  /* time window size */ blockSize)
    || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-2,blockSize-
1,2*blockSize+3*beam)
    || ClkInInterval(clk, volStart, blockSize,blockSize,blockSize+2*beam)
) {
20     cout<<endl; cout.width(4); cout << clk << " TriX:";
//cout<<" X="; COUT_c_p(3,
cout<<triLinX[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
//cout<<" Y="; COUT_c_p(2,
cout<<triLinX[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
//cout<<" Z="; COUT_c_p(2,
25     cout<<triLinX[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
cout<<triLinX[c].results.weightsXYZ[1].Z()); cout<<" out=vox A:";
else
    cout<<" w=1 vox A:";
    if (!triLinX[1].triLinXPipeline[1].results.perPipelineControlFlags-
30     >valid) cout<<"invalid"; else {
        cout<<" X="; COUT_c_p(3, if
(!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].a).X(); );
        cout<<" Y="; COUT_c_p(2, if
(!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].a).Y(); );
35     cout<<" Z="; COUT_c_p(2, if
(!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].a).Z(); );
        }
        if (0 == triLinX[1].results.weightsXYZ[1].Z()) cout<<"\n\t w=0 vox
B:"; else
            cout<<"\n\t out=vox B:";
            if (!triLinX[1].triLinXPipeline[1].results.perPipelineControlFlags-
40     >valid) cout<<"invalid"; else {
                cout<<" X="; COUT_c_p(3, if
(!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].b).X(); );
                cout<<" Y="; COUT_c_p(2, if
(!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].b).Y(); );
45

```

50

55

```

      125
      cout<<" Z=";           COUT_c_p(2, if
5    (!triLinX[c].triLinXPipeline[p].results.perPipelineControlFlags->valid)
cout<<"I"; else cout<<Voxel(triLinX[c].triLinXPipeline[p].b).Z(); );
      cout<-
Z=<<triLinX[1].results.perPipelineControlFlags[1].voxelPosXYZ.Z() << "(Ctrl)";
      }

/*
10   cout<<" I=";
   COUT_p( 0,2,
cout<<triLinX[c].inputs.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
   cout<<"PBF=";
   for (int partialBeamDelay=blockSize/numOfPipelinesPerChip - 1;
partialBeamDelay>= 0; --partialBeamDelay)
   COUT_p( 0,2,
15  cout<<triLinX[c].triLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Peek(
partialBeamDelay).voxelPosXYZ.Z(); );
   cout<<"CDF=";
   for (int communicationDelay=2 - 1; communicationDelay>= 0; --
communicationDelay)
   COUT_p( 0,2,
20  cout<<triLinX[c].triLinXPipeline[p].communicationDelayPerPipelineControlFlagsFiF
o.Peek(communicationDelay).voxelPosXYZ.Z(); );
   cout<<"CR=";
   COUT_p( 0,2,
cout<<triLinX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
25  cout.width(2) ;
cout<<triLinX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z();
   cout<<" CV=";
   cout.width(2) ;
cout<<triLinX[c].readBufferPerPipelineControlFlags.voxelPosXYZ.Z();
   cout.width(2) ;
30  cout<<triLinX[c].communicationPerPipelineControlFlags.voxelPosXYZ.Z();
*/
   //cout<<" a="; cout<<Voxel(triLinX[0].triLinXPipeline[1].a);
   //cout<<" b="; cout<<Voxel(triLinX[0].triLinXPipeline[1].b);
}
} // DebugTriLinX
35

void Cube4::DebugGradY(const int clk)
{
/*
40   int blockBeamDelay = blockSize / numOfPipelinesPerChip;
   int partialBeamsPerBlockSlice = blockSize * blockSize /
numOfPipelinesPerChip;

   DebugImage(5+ 2*blockBeamDelay + 2*partialBeamsPerBlockSlice,
gradY[0].results.perChipControlFlags);

45   for (int c=0; c<numOfChips; ++c) {
   for (int p=0; p<numOfPipelinesPerChip; ++p){
   DebugWritePixel
   (c, p, (unsigned char)
}
50

```

```

126
(0.5 * (255.0 +
           gradY[c].results.gy[p])),0 ,0);
//          (c, p, (unsigned char)
5 // gradY[c].gradYPipeline[p].results.voxel->raw16bit);

}
}
*/
if (21*10<clk && clk<21*10 + 21*14) {
10 cout <<endl<<clk<<"\tGradY: ";
cout << "in: ";
cout << "voxel[2] "<<gradY[0].inputs.voxel[2]<< "   ";
cout << "\tcurrent[2] "<<gradY[0].gradYPipeline[2].current.raw16bit;
cout << "\tlastFiFoOut[2] "<<gradY[0].gradYPipeline[2].lastFiFoOut.raw16bit;
cout << "\tout: ";
15 cout << "gy[2] "<<gradY[0].results.gy[2];
cout << "\tvoxel[2] "<<gradY[0].results.voxel[2];
cout << "\tVolStart "<<gradY[0].results.perChipControlFlags.volumeStart<<
     ":"<<gradZLinY[0].results.perChipControlFlags.volumeStart;
}
20 } // DebugGradY

void Cube4::DebugGradX(const int clk)
{
/*
25     int blockBeamDelay = blockSize / numOfPipelinesPerChip;
     int partialBeamsPerBlockSlice = blockSize * blockSize /
numOfPipelinesPerChip;

     DebugImage(6 + 2*blockBeamDelay + 2*partialBeamsPerBlockSlice,
gradX[0].results.perChipControlFlags);

30     for (int c=0; c<numOfChips; ++c) {
     for (int p=0; p<numOfPipelinesPerChip; ++p){
DebugWritePixel
(c, p, (unsigned char)
(0.5 * (255.0 + gradX[c].results.gy[p])),0 ,0);
//          (c, p, (unsigned char)
35 // gradY[c].gradYPipeline[p].results.voxel->raw16bit);

}
}
*/
40     if (10*21<clk && clk<11*21+5) {
cout <<endl<<clk<<"\tGradX: ";
cout << "in: ";
//cout << "left[2] "<<gradX[0].inputs.voxel[1].raw16bit<< "   ";
//cout << "\tcurrent[2] "<<gradX[0].inputs.voxel[2].raw16bit;
45 //cout << "\tright[2] "<<gradX[0].inputs.voxel[3].raw16bit;
cout << "\tWeights[10] "<<gradX[0].inputs.weightsXYZ[10];
cout << "\tout: ";
cout << "\tWeights[10] "<<gradX[0].results.weightsXYZ[10];
//cout << "gx[2] "<<gradX[0].results.gx[2];
}
50

```

```

127
    //cout << "\tvoxel[2]           " << gradX[0].results.voxel[2];
}
5   if (gradX[0].results.perChipControlFlags.volumeStart)
    cout << endl << "gradX::volumeStart@" << clk << " ";
} // DebugGradX

void Cube4::DebugGradZ(const int clk)
10 {
    DebugImage(gradZ[0].results.perChipControlFlags);
    for (int c=0; c<numOfChips; ++c) {
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            DebugWritePixel
                (c, gradZ[c].results.perPipelineControlFlags[p].voxelPosXYZ,
15      (unsigned char)
                //((gradZ[c].inputs.voxel0[p].raw16bit), 0, 0);
                (255 + gradZ[c].results.gz[p]) / 2, 0, 0);
        }
    }
20   if (gradZ[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "GradZ::volumeStart@inputs@" << clk << " ";
    if (gradZ[0].results.perChipControlFlags.volumeStart)
        cout << endl << "GradZ::volumeStart@results@" << clk << " ";

25   int volStart=19;
    if (volStart+0*(2*8*8*3)<=clk && clk<volStart+10+1*(2*8*8*3)) {
        cout << endl << clk << "\tGradZ: ";
        cout << "in: ";
        cout << " voxel0[1] " << gradZ[0].gradZPipeline[1].delayVoxel0again << "
30   ";
        cout << "\tvoxel1[1] " << *gradZ[0].gradZPipeline[1].inputs.voxell;
        //cout << "\tweightsXYZ[1] " << gradZ[0].inputs.weightsXYZ[1];
        cout << "\tout: ";
        cout << " gz[1] " << *gradZ[0].gradZPipeline[1].results.gz;
        //cout << "\tweightsXYZ[1] " << gradZ[0].results.weightsXYZ[1];
    }
35   }
} // DebugGradZ

void Cube4::DebugGradZLinZ(const int clk)
40 {
    DebugImage(gradZLinZ[0].results.perChipControlFlags);
    for (int c=0; c<numOfChips; ++c) {
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            DebugWritePixel
                (c, gradZLinZ[c].results.perPipelineControlFlags[p].voxelPosXYZ,
45      (unsigned char)
                //((gradZLinZ[c].inputs.voxel0[p].raw16bit), 0, 0);
                (255 + gradZLinZ[c].results.gz[p]) / 2, 0, 0);
        }
    }
50

```

128

```

if
(gradZLinZ[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "GradZLinZ::volumeStart@inputs@" << clk << " ";
if (gradZLinZ[0].results.perChipControlFlags.volumeStart)
    cout << endl << "GradZLinZ::volumeStart@results@" << clk << " ";

int volStart=19;
if (volStart+0*(2*8*8*3)<=clk && clk<volStart+10+1*(2*8*8*3)) {
    cout << endl << clk << "\tGradZLinZ: ";
    cout << "in: ";
    cout << "gz[2] " << gradZLinZ[0].inputs.gz[2]
        << gradZLinZ[0].inputs.perPipelineControlFlags[2].voxelPosXYZ << " ";
    //cout << "\tweightsXYZ[1] " << gradZLinZ[0].inputs.weightsXYZ[2];
    cout << "\tout: ";
    cout << "a[2] " << gradZLinZ[0].gradZLinZPipeline[2].a << " ";
    cout << "b[2] " << gradZLinZ[0].gradZLinZPipeline[2].b;
    cout << "\tgzResult[2] " << gradZLinZ[0].results.gz[2]
        << gradZLinZ[0].results.perPipelineControlFlags[2].voxelPosXYZ;
    cout << "\tweightsXYZ[2] " << gradZLinZ[0].results.weightsXYZ[2];
}
} // DebugGradZLinZ

void Cube4::DebugGradZLinY(const int clk)
{
25    DebugImage(gradZLinY[0].results.perChipControlFlags);
    for (int c=0; c<numOfChips; ++c) {
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            DebugWritePixel
                (c, gradZLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ,
                 (unsigned char)
                 (255 + gradZLinY[0].results.gz[p]) / 2, 0, 0);
        }
    }

    if (gradZLinY[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "GradZLinY::volumeStart@inputs@" << clk << " ";
    if (gradZLinY[0].results.perChipControlFlags.volumeStart)
        cout << endl << "GradZLinY::volumeStart@results@" << clk << " ";

    int volStart=19;
    if (volStart+0*(2*8*8*3)<=clk && clk<volStart+10+1*(2*8*8*3)) {
        cout << endl << clk << "\tGradZLinY: ";
        cout << "in: ";
        cout << "gz[1] " << gradZLinY[0].inputs.gz[1]
            << gradZLinY[0].inputs.perPipelineControlFlags[1].voxelPosXYZ << " ";
        cout << "\tweightsXYZ[1] " << gradZLinY[0].inputs.weightsXYZ[1];
        cout << "\tout: ";
        cout << "gz[1] " << gradZLinY[0].results.gz[1]
            << gradZLinY[0].results.perPipelineControlFlags[1].voxelPosXYZ;
        cout << "\tweightsXYZ[1] " << gradZLinY[0].results.weightsXYZ[1];
    }
} // DebugGradZLinY

```

55

```

5   void Cube4::DebugGradZLinX(const int clk)
6   {
7     /*
8       int blockBeamDelay = blockSize / numOfPipelinesPerChip;
9       int partialBeamsPerBlockSlice = blockSize * blockSize /
10      numOfPipelinesPerChip;
11
12      DebugImage(5 + blockBeamDelay + 2 * partialBeamsPerBlockSlice,
13      gradZLinX[0].results.perChipControlFlags);
14      for (int c=0; c<numOfChips; ++c) {
15        for (int p=0; p<numOfPipelinesPerChip; ++p) {
16          DebugWritePixel
17            (c, p, (unsigned char)
18             (255 + gradZLinX[0].results.gz[p]) / 2, 0, 0);
19        }
20      }
21    */
22    if (3*21<clk && clk<4*21+5) {
23      cout << endl << clk << "\tGradZLinX: ";
24      cout << "in: ";
25      cout << "  gz[14] " << gradZLinX[0].inputs.gz[14] << "    ";
26      cout << "  \tgz[15] " << gradZLinX[0].inputs.gz[15] << "    ";
27      cout << "  \tweightsXYZ[14] " << gradZLinX[0].inputs.weightsXYZ[14];
28      cout << "  \tout: ";
29      cout << "  gz[14] " << gradZLinX[0].results.gz[14];
30    }
31  } // DebugGradZLinX
32
33  void Cube4::DebugShader(const int clk)
34  {
35    int c,p;
36    DebugImage(shader[0].results.perChipControlFlags);
37    ForAll_c_p(DebugWritePixel
38      (c,
39      shader[c].results.perPipelineControlFlags[p].voxelPosXYZ,
40      shader[c].results.shadel[p].a, 0
41      ,0));
42    //((0.5 * (255.0 + shader[c].inputs.gz[p]))));
43    //((0.5 * (255.0 + shader[c].inputs.gy[p]))));
44    //((0.5 * (255.0 + shader[c].inputs.gx[p])));
45    //shader[c].inputs.voxel[p].raw16bit);
46    //((unsigned char) (255 * shader[c].results.shadel[p].r),
47    //((unsigned char) (255 * shader[c].results.shadel[p].g),
48    //((unsigned char) (255 * shader[c].results.shadel[p].b));
49
50    if (shader[0].inputs.perChipControlFlags->volumeStart)
51      cout << endl << "shader::volumeStart@inputs@" << clk << "  ";
52    if (shader[0].results.perChipControlFlags.volumeStart)
53      cout << endl << "shader::volumeStart@results@" << clk << "  ";
54
55

```

```

int volStart=10 + 2*blockSlice + beam + partialBeam;
5   if (ClkInInterval(/* current clock */ clk,
                      /* pipeline start */ volStart,
                      /* beam/slice start */ 1, 1,
                      /* time window size */ blockSize)
      || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-3,blockSize-
1,2*blockSize+3*beam)
10    || ClkInInterval(clk, volStart, blockSize,blockSize,blockSize+2*beam)
     ) {
      cout<<endl; cout.width(4); cout << clk << " Shader:";
      if (!shader[1].shaderPipeline[1].results.perPipelineControlFlags-
>valid) cout<<"invalid"; else {
         cout<<" X="; COUT_c_p{3, if (!shader[c].shaderPipeline[p].RPV)
15   cout<<"I"; else cout<<shader[c].shaderPipeline[p].results.shadel->r; );
         cout<<" Y="; COUT_c_p{2, if (!shader[c].shaderPipeline[p].RPV)
cout<<"I"; else cout<<shader[c].shaderPipeline[p].results.shadel->g; );
         cout<<" Z="; COUT_c_p{2, if (!shader[c].shaderPipeline[p].RPV)
cout<<"I"; else cout<<shader[c].shaderPipeline[p].results.shadel->b; );
20    }

//cout<<" vox";
//cout<<" X="; COUT_c_p{3,
cout<<shader[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
25 //cout<<" Y="; COUT_c_p{2,
cout<<shader[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
//cout<<" Z="; COUT_c_p{2,
cout<<shader[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );

//cout << "in: ";
30 //cout << " voxel[5] << shader[0].inputs.voxel[5].raw16bit << " ";
//cout << "\tWeights[5] << shader[0].inputs.weightsXYZ[5];
//cout << "\tgx[5] << shader[0].inputs.gx[5] << " ";
//cout << "\tgy[5] << shader[0].inputs.gy[5] << " ";
//cout << "\tgz[5] << shader[0].inputs.gz[5] << " ";
//cout << "\tout: ";
35 //cout << " shadel[5]: << shader[0].results.shadel[5];
}
} // DebugShader

void Cube4::DebugDataSync(const int clk)
40 {
  int c,p;
  DebugImage(dataSync[0].results.perChipControlFlags);
  ForAll_c_p(DebugWritePixel
              (c,
dataSync[c].results.perPipelineControlFlags[p].voxelPosXYZ,
45           dataSync[c].results.shadel[p].a, 0
,0));;

  if (dataSync[0].inputs.perChipControlFlags->volumeStart)

```

```

131
    cout << endl << "dataSync::volumeStart@inputs@" << clk
<< " ";
5     if (dataSync[0].results.perChipControlFlags.volumeStart)
        cout << endl << "dataSync::volumeStart@results@" << clk << " ";

    int volStart=17 + 2*blockSlice + beam + partialBeam;
    if (ClkInInterval(/* current clock */ clk,
                      /* pipeline start */ volStart,
10                  /* beam/slice start */ 1, 1,
                      /* time window size */ blockSize)
        || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-3,blockSize-
1,2*blockSize+3*beam)
        || ClkInInterval(clk, volStart, blockSize,blockSize,blockSize+2*beam)
        ) {
15     cout<<endl; cout.width(4); cout << clk << " DataSync:";
        if
        (!dataSync[1].dataSyncPipeline[1].results.perPipelineControlFlags->valid)
        cout<<"invalid"; else {
            cout<<" X="; COUT_c_p(3, if
20            (!dataSync[c].dataSyncPipeline[p].RPV) cout<<"I"; else
            cout<<dataSync[c].dataSyncPipeline[p].results.shadel->r; );
            cout<<" Y="; COUT_c_p(2, if
            (!dataSync[c].dataSyncPipeline[p].RPV) cout<<"I"; else
            cout<<dataSync[c].dataSyncPipeline[p].results.shadel->g; );
            cout<<" Z="; COUT_c_p(2, if
25            (!dataSync[c].dataSyncPipeline[p].RPV) cout<<"I"; else
            cout<<dataSync[c].dataSyncPipeline[p].results.shadel->b; );
            }
            //cout<<" vox";
            //cout<<" X="; COUT_c_p(3,
            cout<<dataSync[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
30            //cout<<" Y="; COUT_c_p(2,
            cout<<dataSync[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
            //cout<<" Z="; COUT_c_p(2,
            cout<<dataSync[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );

            //cout << "in: ";
35            // cout << "shadel[5]: "<<dataSync[0].inputs.shadel[5];
            //cout << "\tout: ";
            //cout << " shadel[5]: "<<dataSync[0].results.shadel[5];
            //cout << "\tWeights[5] "<<dataSync[0].results.weightsXYZ[5];
        }
40    } // DebugDataSync

    void Cube4::DebugComposBuffer(const int clk)
    {
        int c,p;
        DebugImage(composBuff[0].results.perChipControlFlags);
45        ForAll_c_p(DebugWritePixel
                    (c,
composBuff[c].results.perPipelineControlFlags[p].voxelPosXYZ,

```

50

55

```

132
      composBuff[c].results.coxel[p].a,
0, 0););
/*
5   DebugImage(*composBuff[0].inputs.perChipControlFlags);
   ForAll_c_p(DebugWritePixel
      (c,
composBuff[c].inputs.perPipelineControlFlags[p].voxelPosXYZ,
      composBuff[c].inputs.coxel[p].a,
10 0,0));;
*/
   if (composBuff[0].inputs.perChipControlFlags->volumeStart)
      cout << endl << "composBuff::volumeStart@inputs@" << clk << " ";
   if (composBuff[0].results.perChipControlFlags.volumeStart)
      cout << endl << "composBuff::volumeStart@results@" << clk << " ";
15
   int volStart=15 + 3*blockSlice + 3*beam + 3*partialBeam;
   if (ClkInInterval(/* current clock */ clk,
                     /* pipeline start */ volStart,
                     /* beam/slice start */ 0, 0,
                     /* time window size */ 6*beam+blockSlab+3*blockSize)
20
      || ClkInInterval(clk, volStart, blockSize-1, 0, 2*beam)
      || ClkInInterval(clk, volStart, blockSize+blockSize-2, 0, 5*beam+blockSize)
      |||| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1, blockSize-1, 2*beam)
      ) {
      cout<<endl; cout.width(3); cout << clk << " ComposBuff:";
25
      //cout<<" X="; COUT_c_p(3, cout<<composBuff[c].results.coxel[p].r; );
      //cout<<" Y="; COUT_c_p(2, cout<<composBuff[c].results.coxel[p].g; );
      //cout<<" Z="; COUT_c_p(2, cout<<composBuff[c].results.coxel[p].b; );
      //cout<<"\n\t Ctrl: ";
      cout<<" X="; COUT_c_p(3,
25
      cout<<composBuff[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
      cout<<" Y="; COUT_c_p(2,
30
      cout<<composBuff[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
      cout<<" Z="; COUT_c_p(2,
      cout<<composBuff[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
      }
} // DebugComposBuffer
35

void Cube4::DebugComposSelX(const int clk)
{
   int c,p;
   DebugImage(composSelX[0].results.perChipControlFlags);
40
   ForAll_c_p(DebugWritePixel
      (c,
composSelX[c].results.perPipelineControlFlags[p].voxelPosXYZ,
      composSelX[c].results.coxel[p].a, 0
,0));;
45
   //128.0 * (1.0 + composSelX[c].inputs.weightsXYZ[p].X()),
   //128.0 * (1.0 + composSelX[c].inputs.weightsXYZ[p].Y()));

   if (composSelX[0].inputs.perChipControlFlags->volumeStart)
50

```

```

133
cout << endl << "composSelX::volumeStart@inputs@" <<
clk << " ";
if (composSelX[0].results.perChipControlFlags.volumeStart)
    cout << endl << "composSelX::volumeStart@results@" << clk << " ";

int volStart=13 + 2*blockSlice + beam;
if (ClkInInterval(/* current clock */ clk,
                  /* pipeline start */ volStart,
                  /* beam/slice start */ 0, 0,
                  /* time window size */ 2*blockSlice+6*beam)
    ||| ClkInInterval(clk, volStart, blockSize-1, 0, 2*beam)
    ||| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1, blockSize-1, 2*beam)
    ) {
    cout<<endl; cout.width(3); cout << clk << " ComposSelX:";

15 //cout<<" X="; COUT_c_p(3, cout<<composSelX[c].results.coxel[p].r; );
//cout<<" Y="; COUT_c_p(2, cout<<composSelX[c].results.coxel[p].g; );
//cout<<" Z="; COUT_c_p(2, cout<<composSelX[c].results.coxel[p].a; );

//cout<<" I=";
//COUT_p( 0,2,
20 cout<<composSelX[c].inputs.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
//cout<<"PBF=";
//for (int partialBeamDelay=blockSize/numOfPipelinesPerChip - 1;
partialBeamDelay>= 0; --partialBeamDelay)
    // COUT_p( 0,2,
25 cout<<composSelX[c].composSelXPipeline[p].partialBeamPerPipelineControlFlagsFiFo
.Peek(partialBeamDelay).voxelPosXYZ.Z(); );
//cout<<"CDF=";
//for (int communicationDelay=2 - 1; communicationDelay>= 0; --
communicationDelay)
    // COUT_p( 0,2,
30 cout<<composSelX[c].composSelXPipeline[p].communicationDelayPerPipelineControlFl
agsFiFo.Peek(communicationDelay).voxelPosXYZ.Z(); );
//cout<<"CR=";
//COUT_p( 0,2,
cout<<composSelX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
//cout.width(2) ;
35 cout<<composSelX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z();
//cout<<" CV=";
//cout.width(2) ;
cout<<composSelX[c].readBufferPerPipelineControlFlags.voxelPosXYZ.Z();
//cout.width(2) ;
40 cout<<composSelX[c].communicationPerPipelineControlFlags.voxelPosXYZ.Z();

    p=numOfPipelinesPerChip-1;

    cout<<"coixelMiddle[1]:"
"<<*composSelX[0].composSelXPipeline[p].inputs.coixelMiddle
45     <<"
">voxelPosXYZ;
    cout<<"\tcoixelRight[1]:"
"<<*composSelX[0].composSelXPipeline[p].inputs.coixelRight;

```

50

55

```

134
    //cout << "\tWeights[1]
5   "<<*composSelX[0].composSelXPipeline[p].inputs.weightsXYZ;
    cout<< "\tCtrlX[1]
    "<<TypeStr::XStepDirection[composSelX[0].composSelXPipeline[p].inputs.perChipCon
trolFlags->xStep];
    cout<<"\tout: ";
    cout<<"coixel[1]: "<<composSelX[0].results.coixel[p]

10   <<composSelX[0].results.perPipelineControlFlags[p].voxelPosXYZ;
}
} // DebugComposSelX

void Cube4::DebugComposLinX(const int clk)
15 {
    if (composLinX[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "composLinX::volumeStart@inputs@" << clk << " ";
    if (composLinX[0].results.perChipControlFlags.volumeStart)
        cout << endl << "composLinX::volumeStart@results@" << clk << " ";

20    int p;
    int volStart=18 + 2*blockSlice + beam;
    if (ClkInInterval(/* current clock */ clk,
                      /* pipeline start */ volStart,
                      /* beam/slice start */ 0, 0,
25                      /* time window size */ 2*blockSlice+6*beam)
        //|| ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
        //|| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
        ) {
        cout<<endl; cout.width(3); cout << clk << " ComposLinX:";

30    //cout<<" X="; COUT_c_p(3, cout<<composLinX[c].results.coixel[p].r; );
    //cout<<" Y="; COUT_c_p(2, cout<<composLinX[c].results.coixel[p].g; );
    //cout<<" Z="; COUT_c_p(2, cout<<composLinX[c].results.coixel[p].a; );

        //cout<<" I=";
        //COUT_p( 0,2,
    cout<<composLinX[c].inputs.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
35    //cout<<"PBF=";
        //for (int partialBeamDelay=blockSize/numOfPipelinesPerChip - 1;
partialBeamDelay>= 0; --partialBeamDelay)
        // COUT_p( 0,2,
    cout<<composLinX[c].composLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo
        .Peek(partialBeamDelay).voxelPosXYZ.Z(); );
40    //cout<<"CDF=";
        //for (int communicationDelay=2 - 1; communicationDelay>= 0; --
communicationDelay)
        // COUT_p( 0,2,
    cout<<composLinX[c].composLinXPipeline[p].communicationDelayPerPipelineControlFl
agsFiFo.Peek(communicationDelay).voxelPosXYZ.Z(); );
45    //cout<<"CR=";
        //COUT_p( 0,2,
    cout<<composLinX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z(); );

```

50

55

135

```

    //cout.width(2) ;
cout<<composLinX[c].computation.perPipelineControlFlags[p].voxelPosXYZ.Z();
    //cout<<" CV=";
    //cout.width(2) ;
5   cout<<composLinX[c].readBufferPerPipelineControlFlags.voxelPosXYZ.Z();
    //cout.width(2) ;
cout<<composLinX[c].communicationPerPipelineControlFlags.voxelPosXYZ.Z();

10  p=0;

    cout<<"coixelLeft[1]:
*<<*composLinX[0].composLinXPipeline[p].inputs.coixelLeft;
    cout<<"\tcoixelMiddle[1]:
*<<*composLinX[0].composLinXPipeline[p].inputs.coixelMiddle
15  <<
*<<composLinX[0].composLinXPipeline[p].inputs.perPipelineControlFlags-
>voxelPosXYZ;
    //cout << "\tWeights[1]
*<<*composLinX[0].composLinXPipeline[p].inputs.weightsXYZ;
    cout<<"\tCtrlX[1]
20  *<<TypeStr::XStepDirection[composLinX[0].composLinXPipeline[p].inputs.perChipCon-
trolFlags->xStep];
    cout<<"\tout: ";
    cout<<"coixel[1]: "<<composLinX[0].results.coixel[p]

        <<composLinX[0].results.perPipelineControlFlags[p].voxelPosXYZ;
25  }
} // DebugComposLinX

void Cube4::DebugComposSelY(const int clk)
{
30  if (composSelY[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "composSelY::volumeStart@inputs@" << clk << " ";
    if (composSelY[0].results.perChipControlFlags.volumeStart)
    cout << endl << "composSelY::volumeStart@results@" << clk << " ";

35  int c,p;
int volStart=13 + 2*blockSlice + beam;
if (ClkInInterval(/* current clock */ clk,
                  /* pipeline start */ volStart,
                  /* beam/slice start */ 0, 0,
                  /* time window size */ 2*blockSlice+6*beam)
40  //|| ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
    //|| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
    ) {
    cout<<endl; cout.width(3); cout << clk << " ComposSelY:";
    //cout<<" X="; COUT_c_p(3, cout<<composSelY[c].results.coixel[p].r; );
    //cout<<" Y="; COUT_c_p(2, cout<<composSelY[c].results.coixel[p].g; );
45  //cout<<" Z="; COUT_c_p(2, cout<<composSelY[c].results.coixel[p].a; );

    c=0; p=0;
}

```

50

55

```

136
    cout<<"coxFifo[1]: "
5     <<composSelY[c].composSelYPipeline[p].fifoCoxel
        <<
    " <<composSelY[c].composSelYPipeline[p].results.perPipelineControlFlags-
>voxelPosXYZ;
        cout<<"\tcoxFifo[1]: "<<*composSelY[c].composSelYPipeline[p].inputs.coxFoxel
        <<
    " <<composSelY[c].composSelYPipeline[p].inputs.perPipelineControlFlags-
>voxelPosXYZ;
10    //cout << "\tWeights[1]
    " <<*composSelY[c].composSelYPipeline[p].inputs.weightsXYZ;
        cout<< "\tCtrlX[1]
    " <<TypeStr::YStepDirection[composSelY[c].composSelYPipeline[p].inputs.perChipCon-
trolFlags->yStep];
        cout<<"\tout: ";
15    cout<<"coxFifo[1]: "<<composSelY[c].results.coxFoxel[p]

        <<composSelY[c].results.perPipelineControlFlags[p].voxelPosXYZ;
    }
} // DebugComposSelY
20

void Cube4::DebugComposLinY(const int clk)
{
    if (composLinY[0].inputs.perChipControlFlags->volumeStart)
        cout << endl << "composLinY::volumeStart@inputs@" << clk << " ";
25    if (composLinY[0].results.perChipControlFlags.volumeStart)
        cout << endl << "composLinY::volumeStart@results@" << clk << " ";

    int c,p;
    int volStart=13 + 2*blockSlice + beam;
    if (ClkInInterval(/* current clock */ clk,
30        /* pipeline start */ volStart,
        /* beam/slice start */ 0, 0,
        /* time window size */ 2*blockSlice+6*beam)
        //|| ClkInInterval(clk, volStart, blockSize-1,0,2*beam)
        //|| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1,blockSize-1,2*beam)
        ) {
35    cout<<endl; cout.width(3); cout << clk << " ComposLinY:";

    //cout<<" X="; COUT_c_p(3, cout<<composLinY[c].results.coxFoxel[p].r);
    //cout<<" Y="; COUT_c_p(2, cout<<composLinY[c].results.coxFoxel[p].g);
    //cout<<" Z="; COUT_c_p(2, cout<<composLinY[c].results.coxFoxel[p].a);

40    c=0; p=0;

    cout<<"coxFifo[1]: "<<composLinY[c].composLinYPipeline[p].fifoCoxFoxel
        <<
    " <<composLinY[c].composLinYPipeline[p].results.perPipelineControlFlags-
>voxelPosXYZ;
        cout<<"\tcoxFifo[1]: "<<*composLinY[c].composLinYPipeline[p].inputs.coxFoxel
        <<
    " <<composLinY[c].composLinYPipeline[p].inputs.perPipelineControlFlags-
>voxelPosXYZ;
45
50

```

137

```

    //cout << "\tWeights[1]
" << *composLinY[c].composLinYPipeline[p].inputs.weightsXYZ;
    cout << "\tCtrlX[1]
5   " << TypeStr::YStepDirection[composLinY[c].composLinYPipeline[p].inputs.perChipCon
trolFlags->yStep];
    cout << "\tout: ";
    cout << "coixel[1]: " << composLinY[c].results.coixel[p]

    << composLinY[c].results.perPipelineControlFlags[p].voxelPosXYZ;
10
} // DebugComposLinY

void Cube4::DebugCompos(const int clk)
15 {
    int c,p;
    DebugImage(compos[0].results.perChipControlFlags);
    ForAll_c_p(DebugWritePixel
                (c,
compos[c].results.perPipelineControlFlags[p].voxelPosXYZ,
20
                compos[c].results.coixel[p].a, 0 ,0));
    // (255 * compos[c].results.coixel[p].r),
    // (255 * compos[c].results.coixel[p].g),
    // (255 * compos[c].results.coixel[p].b);

    if (compos[0].inputs.perChipControlFlags->volumeStart)
25      cout << endl << "compos::volumeStart@inputs@" << clk << "  ";
    if (compos[0].results.perChipControlFlags.volumeStart)
      cout << endl << "compos::volumeStart@results@" << clk << "  ";

    if (dataSync[0].results.perChipControlFlags.volumeStart)
30      cout << endl << "dataSync::volumeStart@results@" << clk << "  ";
    if (composBuff[0].results.perChipControlFlags.volumeStart)
      cout << endl << "composBuff::volumeStart@results@" << clk << "  ";

    //int volStart=14 + 2*blockSlice + 3*beam + 3*partialBeam;
    int volStart=17 + 2*blockSlice + beam + partialBeam;
35    if (ClkInInterval(/* current clock */ clk,
                      /* pipeline start */ volStart,
                      /* beam/slice start */ 1, 1,
                      /* time window size */ blockSize)
        || ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-3,blockSize-
1,2*blockSize+3*beam)
        || ClkInInterval(clk, volStart, blockSize,blockSize,blockSize+2*beam)
40    ) {
        cout<<endl; cout.width(4); cout << clk << " Compos: shadel:";
        if (!compos[1].composPipeline[1].results.perPipelineControlFlags-
>valid) cout<<"invalid"; else {
            cout<<" X="; COUT_c_p(3, if (!compos[c].composPipeline[p].RPV)
45            cout<<" I"; else cout<<compos[c].composPipeline[p].inputs.shadel->r; );
            cout<<" Y="; COUT_c_p(2, if (!compos[c].composPipeline[p].RPV)
            cout<<" I"; else cout<<compos[c].composPipeline[p].inputs.shadel->g; );
}

```

50

55

```

138
    cout<<" Z="; COUT_c_p(2, if
5      (!compos[c].composPipeline[p].RPV) cout<<"I"; else
      cout<<compos[c].composPipeline[p].inputs.shadel->b; );
    }
    cout<<"\n\t coxel:";
    if (!compos[1].composPipeline[1].results.perPipelineControlFlags-
>valid) cout<<"invalid"; else {
      cout<<" X="; COUT_c_p(3, if (!compos[c].composPipeline[p].RPV)
10     cout<<"I"; else cout<<compos[c].composPipeline[p].inputs.coxel->r; );
      cout<<" Y="; COUT_c_p(2, if (!compos[c].composPipeline[p].RPV)
      cout<<"I"; else cout<<compos[c].composPipeline[p].inputs.coxel->g; );
      cout<<" Z="; COUT_c_p(2, if (!compos[c].composPipeline[p].RPV)
      cout<<"I"; else cout<<compos[c].composPipeline[p].inputs.coxel->b; );
    }
15

//cout<<" vox";
//cout<<" X="; COUT_c_p(3,
cout<<compos[c].results.perPipelineControlFlags[p].voxelPosXYZ.X(); );
//cout<<" Y="; COUT_c_p(2,
20  cout<<compos[c].results.perPipelineControlFlags[p].voxelPosXYZ.Y(); );
//cout<<" Z="; COUT_c_p(2,
cout<<compos[c].results.perPipelineControlFlags[p].voxelPosXYZ.Z(); );
//cout<<" X="; COUT_c_p(3, cout<<compos[c].results.coxel[p].r; );
//cout<<" Y="; COUT_c_p(2, cout<<compos[c].results.coxel[p].g; );
25  //cout<<" Z="; COUT_c_p(2, cout<<compos[c].results.coxel[p].a; );

//cout << "in: ";
//      cout << "shadel[5] "<<compos[0].inputs.shadel[5] << "";
//      cout << "coxel[5] "<<compos[0].inputs.coxel[5] << "";
//cout << "\tWeights[5] "<<compos[0].inputs.weightsXYZ[5];
30  //cout << "\tvoxPos[5] " << voxelPos[5];

//cout << "\tout: ";
//      cout << "coxel[5]: "<<compos[0].results.coxel[5];
}
} // DebugCompos
35

void Cube4::DebugFinalCoxelBuffer(const int clk)
{
  int c,p;
  DebugImage(*finalCoxelBuff[0].inputs.perChipControlFlags);
40  ForAll_c_p(DebugWritePixel
              (c,
finalCoxelBuff[c].inputs.perPipelineControlFlags[p].voxelPosXYZ,
              finalCoxelBuff[c].inputs.coxel[p].a,
0,0));
45  if (finalCoxelBuff[0].inputs.perChipControlFlags->volumeStart)
    cout << endl << "finalCoxelBuff::volumeStart@inputs@" << clk << " ";
  int volStart=21 + 3*blockSlice + beam;
50

```

```

139
if (ClkInInterval(/* current clock */ clk,
                  /* pipeline start */ volStart,
                  /* beam/slice start */ 0, 0,
                  /* time window size */ 6*beam)
    ||| ClkInInterval(clk, volStart, blockSize-1, 0, 2*beam)
    ||| ClkInInterval(clk, volStart, datasetSizeXYZ.Y()-1, blockSize-1, 2*beam)
) {
    cout<<endl; cout.width(3); cout << clk << " ComposBuff:";
    cout<<" X="; COUT_c_p(3, cout<<finalCoxelBuff[c].inputs.coxel[p].r; );
    cout<<" Y="; COUT_c_p(2, cout<<finalCoxelBuff[c].inputs.coxel[p].g; );
    cout<<" Z="; COUT_c_p(2, cout<<finalCoxelBuff[c].inputs.coxel[p].a; );
}
} // DebugFinalCoxelBuffer

15
void Cube4::DebugImage(const PerChipControlFlags & flags)
{
    // sliceDelay is number of silice buffers in pipeline up to current stage
    int z;
    static int volumeStartCounter(0);
    static bool first(true), writeImageVolume(false);
    static char fileNameRoot[100], rgbFileName[110], systemCallStr[200];

    if (first) {
        system("/usr/bin/rm -f img/voxelSlc*.miff \n");
        system("/usr/bin/rm -f img/voxelSlc*.rgb \n");
        sliceSize = datasetSizeXYZ.X() * datasetSizeXYZ.Y();
        volumeSize = sliceSize * datasetSizeXYZ.Z();
        slabSize = sliceSize * blockSize;
        x_step = 1;
        y_step = datasetSizeXYZ.X();
        z_step = 3*sliceSize; //rgb
        volumeBuffer = new unsigned char [3*(volumeSize+2*slabSize)]; //rgb
        first = false;
    }

    // write volume image one silice of blocks (slab)
    // after end of volume control bit
    // at that point all blocks of the main volume are guaranteed to be
35    finished

    if (volumeStartCounter==2 && flags.zBlockStart) writeImageVolume = true;
    else                                         writeImageVolume = false;
40    if (flags.volumeStart)
        ++volumeStartCounter;

    if (writeImageVolume) {
        for (z=0; z<datasetSizeXYZ.Z(); ++z) {
            if (z == datasetSizeXYZ.Z()-1 || !singleImagePerFrameMode) {
                // save slices as rgb image
                sprintf(fileNameRoot,"img/voxelSlc%02i",z);

                sprintf(rgbFileName,"%s.rgb",fileNameRoot);

```

```

140
{
    ofstream voxelSlcRGBFile(rgbFileName);
5     voxelSlcRGBFile.write(&volumeBuffer[3*z*sliceSize],3*sliceSize);
    }
    sprintf(systemCallStr, "%s %ix%i %s.rgb %s.miff; %s
@s.rgb & \n",
10    size",
datasetSizeXYZ.Y(),
    "convert -sample 900%x900% -
datasetSizeXYZ.X(),
fileNameRoot, fileNameRoot,
"/usr/bin/rm -f", fileNameRoot);
15    system(systemCallStr);
}
}

// show image animation
if (!singleImagePerFrameMode)
    system("animate img/voxelSlc*.miff & \n");

20 // show final image of sequence
sprintf(systemCallStr,"%s %s.miff %s ; %s \n",
        "convert -sample 200%x200% ",
        fileNameRoot, "img/baseplaneImage.miff",
        ""); //display
25 img/baseplaneImage.miff & ");
    system(systemCallStr);

    // Stop main computation loop
    clk = -10;
30 }
} // DebugImage

void Cube4::DebugWritePixel(const int c,
35             const Vector3D<int> & voxelPosXYZ,
             const unsigned char red,
             const unsigned char green,
             const unsigned char blue)
{
    int r=0,g=0,b=0;
    static const int tile(190);
40
    r = (c==4)*tile + (c==5)*tile + (c==6)*tile;
    g = (c==1)*tile + (c==3)*tile + (c==5)*tile;
    b = (c==2)*tile + (c==3)*tile + (c==6)*tile;

    r += red;
    g += green;
45    b += blue;

    if (r>255) r=255;
50
}

```

141

```

if (g>255) g=255;
if (b>255) b=255;

5   // write Compos results into slab buffer
int x,y,z;

x = voxelPosXYZ.X() * x_step;
y = voxelPosXYZ.Y() * y_step;
z = voxelPosXYZ.Z() * z_step;

volumeBuffer[ x+y+z + 0*sliceSize ] = r;
volumeBuffer[ x+y+z + 1*sliceSize ] = g;
volumeBuffer[ x+y+z + 2*sliceSize ] = b;
} // DebugWritePixel

15

// end of Cube4Debug.C
:::::::::::
cube4/Cube4Interface.C
20 :::::::::::
// Cube4Interface.C
// (c) Ingmar Bitter '97 / Urs Kanus '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

25 void Cube4::Demo()
{
    Cube4 cube4;
    cout << endl << "Demo of class " << typeid(cube4).name();
    cout << endl << "size : " << sizeof(Cube4) << " Bytes";
30    cout << endl << "public member functions:";
    cout << endl << "    cube4.SetDatasetFile()      = " <<
cube4.SetDatasetFile("go");
    cout << endl << "    cube4.SetColorTableFile() = " <<
cube4.SetColorTableFile("go");
35    cout << endl << "    cube4.SetReflectanceTableFile() = " <<
cube4.SetReflectanceTableFile("go");
    cout << endl << "    cube4.SetImageFileRoot()  = " <<
cube4.SetImageFileRoot("go");
    cout << endl << "    cube4.LoadColorTable()     =," << cube4.LoadColorTable();
    cout << endl << "    cube4.LoadAlphaTable()    =," << cube4.LoadAlphaTable();
40    cout << endl << "    cube4.LoadReflectanceTable() =," <<
cube4.LoadReflectanceTable();
    cout << endl << "    cube4.PrintUsage()        =," << cube4.PrintUsage();
    cout << endl << "    cube4.RenderImage()       =," << cube4.RenderImage();
    cout << endl << "End of demo of class " << typeid(cube4).name() << endl;
} // Demo
45

```

```
///////////////
```

50

55

```

142
// show/set data & data properties
//
5 // - local show/set functions

    bool Cube4::PrintUsage() const
    {
        cout << endl << "Usage: cube4 [-ds <dataset file>] [-ct <colorTable
10 file>]" << endl;
        cout << "\t [-at <alpha table file>] [-rt <reflectance table file>]" <<
endl;
        cout << "\t [-ir <image file root>]" << endl;
        cout << "\t [-nc <num of chips>] [-np <num of pipelines>] [-pa][-pe]" <<
endl;
15        cout << "\t [-bs <blocksize>] [-tm \"<image matrix>\"]" << endl;
        cout << "\t [-th <levoy class. threshold>] [-mw <levoy class.
maxwidth>]" << endl;
        cout << "\t [-si] [-bf] [-fb] [-an <num of frames in animation>] [-li
\ "<light vector>\"]" << endl;
        cout << "\t [-px <Phong exponent>] [-ip \"<Pointlight
20 intensity>\"]" << endl;
        cout << "\t [-ia \"<Ambient intensity>\"] [-ka <Ambient coefficient>]" << endl;
        cout << "\t [-kd <Diffuse coefficient>] [-ks <Specular coefficient>] [-sp]
[-sm] [-sv]" << endl;
        cout << "\t [-sc <Shadel color bits>] [-sa <Shadel alpha bits>]" << endl;
        cout << "\t [-c4] [-cl] [-cc <Coxel color bits>] [-ca <Coxel alpha
25 bits>]" << endl;
        cout << "\t [-tw <Trilin weight bits>] [-gc <Gradient component bits>]" <<
endl;
        cout << "\t [-ni] [-ar][-nd][-rs <Reflectance map address bits>]" << endl;
        cout << "\t [-ts <Atan table address bits>] [-cs <Color xfer table address
30 bits>]" << endl;
        cout << "\t [-as <Alpha xfer table address bits>]" << endl;
        exit(0);
    }

    return true;
} // PrintUsage
35

    bool Cube4::SetDatasetFile(const char * fileName)
{
    char str[200];

40    // copy string starting with filename
    strcpy(str, fileName);

    // terminate filename at first space
    str[strcspn(fileName, " ")] = 0;

45    // compare to previous dataset filename
    if (datasetLoaded) datasetLoaded = (strcmp(str, datasetFileName) == 0);

    // copy new filename into class variable
50
}

```

55

```

143
    if (!datasetLoaded)           strcpy(datasetFileName, str);

5     return strlen(datasetFileName) > 0;
} // SetDatasetFile

10
bool Cube4::SetColorTableFile(const char * fileName)
{
11     char str[200];

12     // copy string starting with filename
13     strcpy(str, fileName);

14     // terminate filename at first space
15     str[strcspn(fileName, " ")] = 0;

16     // compare to previous colortable filename
17     if (colorTableLoaded) colorTableLoaded = (strcmp(str, colorTableName)
18 == 0);

19     // copy new filename into class variable
20     if (!colorTableLoaded) strcpy(colorTableName, str);

21     return strlen(colorTableName) > 0;
} // SetColorTableFile

25

bool Cube4::SetAlphaTableFile(const char *.fileName)
{
26     char str[200];

27     // copy string starting with filename
28     strcpy(str, fileName);

29     // terminate filename at first space
30     str[strcspn(fileName, " ")] = 0;

31     // compare to previous alphatable filename
32     if (alphaTableLoaded) alphaTableLoaded = (strcmp(str, alphaTableName)
33 == 0);

34     // copy new filename into class variable
35     if (!alphaTableLoaded) strcpy(alphaTableName, str);

36     return strlen(alphaTableName) > 0;
} // SetAlphaTableFile

40

45     bool Cube4::SetReflectanceTableFile(const char * fileName)
{
46         char str[200];

47         // copy string starting with filename

```

50

55

```

144
    strcpy(str, fileName);

5     // terminate filename at first space
    str[strcspn(fileName, " ")] = 0;

    // compare to previous illumtable filename
    if (reflectanceTableLoaded) reflectanceTableLoaded = (strcmp(str,
reflectanceTableName) == 0);

10    // copy new filename into class variable
    if (!reflectanceTableLoaded) strcpy(reflectanceTableName, str);

    return strlen(reflectanceTableName) > 0;
15 } // SetReflectanceTableFile

bool Cube4::SetImageFileRoot(const char * fileName)
{
20    // copy string starting with filename
    strcpy(imageRootFileName, fileName);

    // terminate filename at first space
    imageRootFileName[strcspn(fileName, " ")] = 0;

25    return strlen(imageRootFileName) > 0;
} // SetImageFileRoot

bool Cube4::LoadColorTable()
30 {
    return true;
} // LoadColorTable

bool Cube4::LoadAlphaTable()
35 {
    return true;
} // LoadAlphaTable

40 bool Cube4::LoadReflectanceTable()
{
    return true;
} // LoadReflectanceTable

45 //////////////////////////////////////////////////////////////////
// internal utility functions

50 bool Cube4::ReadCube4PipelineParametersFromFile(const char * srcFileName)
{
    ifstream srcFile(srcFileName);

```

55

```

char line[256];
srcFile.getline(line,255);
5   while (srcFile) {
        SetParameter(&line[0], &line[4]);
        srcFile.getline(line,255);
    }
    srcFile.close();
10  return true;
} // ReadCube4PipelineParametersFromFile

bool Cube4::ReadCube4PipelineParametersFromCommandLine(int argc, char *argv[])
{
15    if (!argc) // no command line parameters (not even cube4 itself)
        // => internally created Cube4 instance
        return false;

    int pos(0);
20    while (++pos < argc) {
        if (strncmp(argv[pos],"-h",2)==0) { PrintUsage(); }
        else SetParameter(argv[pos], argv[Min(argc-1, pos+1)]);
    }

    return true;
25 } // ReadCube4PipelineParametersFromCommandLine

bool Cube4::SetParameter(const char * id, const char * str)
{
30    // all parameters are assumed to have exactly 2 specifying characters

    // terminate parameter at first space
    // (parameter file might have description afterwards)
    char param[200];
    strcpy(param, str);
    param[strcspn(str, " ")] = 0;

35    // skip some typing :
#define CMP(sw)      (strncmp(id,(sw),3)==0)
#define PRN(strA,strB) cout << endl << "switch -" << (strA) << (strB);

    if      CMP("-pf") { PRN("pf: using parameter file: ",param);
40    ReadCube4PipelineParametersFromFile(param); }
        else if CMP("-ds") { PRN("ds: using dataset: ",param);
        SetDatasetFile(param); }
        else if CMP("-ct") { PRN("ct: using colorTable: ",param);
45    SetColorTableFile(param); }
        else if CMP("-at") { PRN("at: using alphaTable: ",param);
        SetAlphaTableFile(param); }
        else if CMP("-rt") { PRN("rt: using ReflectanceTable: ",param);
        SetReflectanceTableFile(param); }

50

```

```

146
    else if CMP("-ir") { PRN("ir:      using image root: ",param);
    SetImageFileRoot(param); }
5     else if CMP("-nc") { PRN("nc: number of cube4 chips:",param); numOfChips =
atoi(param); }
        else if CMP("-np") { PRN("np: number of pipelines per chip:",param);
numOfPipelinesPerChip = atoi(param); }
            else if CMP("-bs") { PRN("bs: block size:",param);
                blockSize = atoi(param); }
                else if CMP("-pa") { PRN("pa: parallel projection","");
projectionStyle = parallel; }
                    else if CMP("-pe") { PRN("pe: perspective projection","");
projectionStyle = perspective; }
                        else if CMP("-vp") { PRN("vp: view point","");
SetViewPoint(str) ; cout <<
viewPointUVW; }
15     else if CMP("-tm") { PRN("tm: global dataset->image matrix:","");
SetDatasetToImageMatrix(str); }
        else if CMP("-th") { PRN("th: classification threshold: ",param);
threshold = atof(param);}
            else if CMP("-mw") { PRN("mw: classification maxwidth: ",param);
maxWidth = atof(param); }
20     else if CMP("-ma") { PRN("ma: classification alpha multiplyer: ",param);
mulAlpha = atof(param); }
        else if CMP("-si") { PRN("si: single image per frame","");
singleImagePerFrameMode = true; }
            else if CMP("-bf") { PRN("bf: BackToFront compositing","");
compositingStyle = BackToFront; }
25     else if CMP("-fb") { PRN("fb: FrontToBack compositing","");
compositingStyle = FrontToBack; }
            else if CMP("-an") { PRN("an: animation (render multiple frames)",param);
numOfFramesInAnimation = atoi(param); }
                else if CMP("-li") { PRN("li: light source","");
lightSourceUVW.SetSize(lightSourceUVW.Size()+1);
30
    lightSourceUVW[lightSourceUVW.Size()-1](str) ;
                                cout <<
" << lightSourceUVW[lightSourceUVW.Size()-1];
        else if CMP("-px") { PRN("px: Phong exponent: ", param); phongExponent =
atof(param); }
35     else if CMP("-ia") { PRN("ia: Phong ambient intensity: ", param);
SetAmbientIntensity(str) ; }
            else if CMP("-ka") { PRN("ka: Phong ambient coefficient: ", param);Ka =
atof(param); ; }
                else if CMP("-ks") { PRN("ks: Phong specular coefficient: ", param); Ks =
atof(param); ; }
40     else if CMP("-kd") { PRN("kd: Phong diffuse coefficient: ", param); Kd =
atof(param); ; }
                else if CMP("-so") { PRN("so: Shading off (no shading at all): ",param);
shaderMode = NoShading; }
                    else if CMP("-sp") { PRN("sp: Phong shading mode","");
shaderMode = Phong; }
45     else if CMP("-sm") { PRN("sm: Reflectance map shading mode","");
shaderMode = RefMap; }
            else if CMP("-sv") { PRN("sv: Reflectance vector shading mode","");
shaderMode = RefVector; }

```

50

55

```

147
    else if CMP("-sc") { PRN("sc: Shadel color bits: ", param); shadelColorBits
5     = (char)atoi(param); }
    else if CMP("-sa") { PRN("sa: Shadel alpha bits: ", param); shadelAlphaBits =
(char)atoi(param); }
    else if CMP("-c4") { PRN("c4: Cube4 classic mode", ""); cubeMode =
Cube4Classic; }
    else if CMP("-cl") { PRN("cl: Cube4 light mode", ""); cubeMode = Cube4Light; }
    else if CMP("-cc") { PRN("cc: Coxel color bits: ", param); coxelColorBits =
10   (char)atoi(param); }
    else if CMP("-ca") { PRN("ca: Coxel alpha bits: ", param); coxelAlphaBits =
(char)atoi(param); }
    else if CMP("-tw") { PRN("tw: Trilin weight bits: ", param); trilinWeightBits =
composWeightBits = (char)atoi(param); }
    else if CMP("-gc") { PRN("gc: Gradient component bits: ", param); gradientBits
15   = atoi(param); }
    else if CMP("-rw") { PRN("rw: Reflectance map weight bits: ", param);
mapWeightBits = atoi(param); }
    else if CMP("-ni") { PRN("ni: Reflectance map with no interpolation", "");
reflectanceMapIP = false; }
    else if CMP("-rs") { PRN("rs: Reflectance map Size (address bits): ", param);
reflectanceMapBits = atoi(param); }
    else if CMP("-nd") { PRN("nd: Reflectance map no distortion compensation "
"); distortionCompensation = false; }
    else if CMP("-ar") { PRN("ar: Reflectance map constant angular resolution ",
"); constantAngularResolution = true; }
    else if CMP("-cs") { PRN("cs: Color transferfunction table size (address
25   bits): ", param); colorTableBits = atoi(param); }
    else if CMP("-as") { PRN("as: Alpha transferfunction table size (address
bits): ", param); alphaTableBits = atoi(param); }

        return true;
30 } // SetParameter

```

```

35 bool Cube4::SetAmbientIntensity(const char * str)
{
    char param[200];
    strstream ambientIntensityData;
    double tempX, tempY, tempZ;

40    if (str[0] == '') {
        // ambientIntensity from command-file

        // copy only part between double quotes
        strcpy(param, &str[1]);
        param[strcspn(param, "\r\n")] = 0;
45        ambientIntensityData << param;
    }
    else {
        // ambientIntensity from command-line

```

50

55

```

      // copy only part between double quotes
5      ambientIntensityData << str;
    }
    ambientIntensityData>>tempX;
    ambientIntensityData>>tempY;
10   ambientIntensityData>>tempZ;

IAmbient(tempX, tempY, tempZ);

cout << " <<IAmbient<< , ";
15   return true;

} // SetAmbientIntensity

bool Cube4::SetViewPoint(const char * str)
{
20
    char param[200];
    strstream viewPointData;
    double tempX, tempY, tempZ;

    if (str[0] == '\"') {
        // viewPoint from command-file

        // copy only part between double quotes
        strcpy(param, &str[1]);
        param[strcspn(param, "\"")] = 0;
30        viewPointData << param;
    }
    else {
        // viewPoint from command-line

        // copy only part between double quotes
        viewPointData << str;
    }
    viewPointData>>tempX;
    viewPointData>>tempY;
40    viewPointData>>tempZ;

    viewPointUVW(tempX, tempY, tempZ);
    viewPointSet = true;
45
    return true;
} // SetViewPoint

50
bool Cube4::SetDatasetToImageMatrix(const char * str)
{
    char param[200];
    strstream matrixData;

```

```

5      if (str[0] == '') {
          // matrix is given explicitly

          // copy only part between double quotes
          strcpy(param, &str[1]);
          param[strcspn(param, "\r\n")] = 0;

10     matrixData << param;
    }
else {
          // matrix from command-line

15     // copy only part between double quotes

          matrixData << str;
    }

20     double matrix[4][4];
for (int j=0; j<4; ++j) {
          for (int k=0; k<4; ++k) {
              matrixData >> matrix[j][k];
              cout << " " << matrix[j][k];
        }
        if (j<3) cout << ",";
    }

    return true;
} // SetDatasetToImageMatrix
30

void Cube4::GetFinalImage()
{
    // copy final coxel data into rgb memory array

35    int X = datasetSizeXYZ.X();
    int Y = datasetSizeXYZ.Y();
    int Z = datasetSizeXYZ.Z();
    int sliceSize = X*Y;
40    unsigned char * backImageRGB = new unsigned char [3 * sliceSize];
    unsigned char * toBoImageRGB = new unsigned char [3 * sliceSize];
    unsigned char * sideImageRGB = new unsigned char [3 * sliceSize];

    // FinalCoxelBuffer::SideAdrr();

45    int indexR, indexG, indexB, oneFaceIndex, x, y, address, color;
    indexR = 0 * sliceSize;
    indexG = 1 * sliceSize;
    indexB = 2 * sliceSize;
    for (y=0; y<Y; ++y) {
        for (x=0; x<X; ++x) {
            address = y*X + x;
50
55

```

```

150
    backImageRGB[indexR] = (unsigned char) (coxMem[0](address).r *
* 255.0);
5     backImageRGB[indexG] = (unsigned char) (coxMem[0](address).g *
255.0);
         backImageRGB[indexB] = (unsigned char) (coxMem[0](address).b *
255.0);
         ++indexR; ++indexG; ++indexB;
     }
10
}
indexR = 0*sliceSize;
indexG = 1*sliceSize;
indexB = 2*sliceSize;
for (y=0; y<Y; ++y) {
    for (x=0; x<X; ++x) {
        address = sliceSize + y*X + x;
15     toBoImageRGB[indexR] = (unsigned char) (coxMem[0](address).r *
255.0);
         toBoImageRGB[indexG] = (unsigned char) (coxMem[0](address).g *
255.0);
         toBoImageRGB[indexB] = (unsigned char) (coxMem[0](address).b *
20     255.0);
         ++indexR; ++indexG; ++indexB;
     }
}
25
indexR = 0*sliceSize;
indexG = 1*sliceSize;
indexB = 2*sliceSize;
for (y=0; y<Y; ++y) {
    for (x=0; x<X; ++x) {
        address = 2*sliceSize + y*X + x;
30     sideImageRGB[indexR] = (unsigned char) (coxMem[0](address).r *
255.0);
         sideImageRGB[indexG] = (unsigned char) (coxMem[0](address).g *
255.0);
         sideImageRGB[indexB] = (unsigned char) (coxMem[0](address).b *
255.0);
         ++indexR; ++indexG; ++indexB;
35
}
}

///////////////////////////////
// recompute sightray as done in control unit

40 Vector3D<FixPointNumber> sightRay, normalizedSightRay, samplePosIncrement;
sightRay = datasetSizeXYZ;
sightRay /= 2.0;
sightRay -= viewPointXYZ;

45 // normalize by z component
FixPointNumber norm(sightRay.Z().Abs());
normalizedSightRay(sightRay.X() / norm,

```

50

55

151

sightRay.Y() /

norm,

5

```
if (compositingStyle == FrontToBack)
    samplePosIncrement = normalizedSightRay;
else if (compositingStyle == BackToFront)
    samplePosIncrement = -normalizedSightRay;
```

10

```
double dx = double(samplePosIncrement.X());
double dy = double(samplePosIncrement.Y());
double dxAbs = ABS(dx);
double dyAbs = ABS(dy);
```

15

```
int xOffset = int(dxAbs * (Z-1.0)); // -1 for 0 offset start at first slice
```

```
int yOffset = int(dyAbs * (Z-1.0));
```

20

```
// make one big baseplane image
unsigned char * baseplaneImageRGB = new unsigned char [3 * 9*sliceSize];
for (x=0; x<3 * 9*sliceSize; ++x) baseplaneImageRGB[x]=0;
int baseplaneIndex = 9*sliceSize;
```

25

```
// back face
```

```
oneFaceIndex = 0;
for (color=0; color<3; ++color) {
```

```
    for (y=Y; y<2*Y; ++y) {
        for (x=X; x<2*X; ++x) {
```

```
            baseplaneImageRGB[baseplaneIndex*color + y*3*X + x]
                = backImageRGB[oneFaceIndex++];
        }
    }
}
```

30

```
// top/bottom face
```

```
if (yOffset>0) {
    oneFaceIndex = 0;
    for (color=0; color<3; ++color) {
```

```
        for (y=0; y<=yOffset; ++y) {
            for (x=0; x<X; ++x) {

```

```
                baseplaneImageRGB[baseplaneIndex*color
                    + (y+Y-
                        yOffset-1)*3*X
                    + X+x-]
```

40

```
int((yOffset+1-y)*(dxAbs/dyAbs)))
    = toBoImageRGB[oneFaceIndex++];
```

```
}
```

```
} oneFaceIndex += X*(Y-yOffset-1);
```

```
}
```

45

```
// side face
```

```
if (xOffset>0) {
    oneFaceIndex = 0;
```

50

55

174

```

152
for (color=0; color<3; ++color) {
    for (x=0; x<xOffset; ++x) {
        for (y=0; y<Y; ++y) {
            baseplaneImageRGB[baseplaneIndex*color
+ (Y+y-
int((xOffset-1-x)*(dyAbs/dxAbs)))*3*X
+ x+x-
xOffset]
                = sideImageRGB[oneFaceIndex++];
        }
        oneFaceIndex += X*(X-xOffset);
    }
}

15 // write rgb memory array to disk
{
    ofstream backImageRGBfile("img/baseplaneImage.rgb");
    backImageRGBfile.write(baseplaneImageRGB, 3 * 9*sliceSize);
}

20 // convert rgb image to miff image
char str[200];
sprintf(str, "convert -size %ix%i %s %s ; %s \n",
       3*datasetSizeXYZ.X(), 3*datasetSizeXYZ.Y(),
       "img/baseplaneImage.rgb",
25   "img/baseplaneImage.miff",
       "/usr/bin/rm -f img/baseplaneImage.rgb");
system(str);
sprintf(str, "convert -crop %ix%i+%i+%i %s %s \n",
       X+xOffset,Y+yOffset,X-xOffset,Y-yOffset,
       //2*X,2*Y,0,0,
       "img/baseplaneImage.miff",
30   "img/baseplaneImage.miff");
system(str);
sprintf(str, "convert -sample 200%x200% %s %s%s %s \n",
       "img/baseplaneImage.miff", "img/",
imageRootFileName, ".miff");
35 system(str);
// free memory
delete backImageRGB; backImageRGB = 0;
delete toBoImageRGB; toBoImageRGB = 0;
delete sideImageRGB; sideImageRGB = 0;
delete baseplaneImageRGB; baseplaneImageRGB = 0;
40 } // GetFinalImage

void Cube4::MakePSfile(const char * comment)
{
45     char str[500];
{
    ofstream imgInfo("img/imgInfo.tex");
    imgInfo << "\\resizebox{8in}{!}{\\includegraphics{img/"
        << imageRootFileName << ".eps}}" << endl;
}

```

50

55

```

153
    imgInfo << endl << "{\\bf " << imageRootFileName << "}" << endl;
    imgInfo << endl << "{\\it " << comment << ")" << endl;
    imgInfo << endl << "cube mode:-" << TypeStr::CubeMode{cubeMode} <<
5      " ; ";
    imgInfo << endl << "dataset size:-" << datasetSizeUVW << " ; ";
    imgInfo << endl << "view point:-" << viewPointUVW << " ; ";
    imgInfo << endl << "projection style:-"
                           << TypeStr::ProjectionStyle{projectionStyle}
10     << " ;\\\\\\ ";
    imgInfo << endl << "trilin weight bits:-" << trilinWeightBits;
    imgInfo << endl << "gradient component bits:-" << gradientBits <<
" ;\\\\\\ ";
    imgInfo << endl << "shading mode:-"
                           << TypeStr::ShaderMode{shaderMode} << " ; ";
15    imgInfo << endl << "Ka:-" << Ka << " Kd:-" << Kd << " Ks:-" << Ks << " ; ";
    imgInfo << endl << "phong exponent:-" << phongExponent << " ;\\\\\\ ";
    imgInfo << endl << "lights: " << lightSourceUVW << " ;\\\\\\ ";
    imgInfo << endl << "classification: threshold:-" << threshold;
    imgInfo << endl << "maxWidth:-" << maxWidth;
    imgInfo << endl << "mulAlpha:-" << mulAlpha;
20    imgInfo << endl << "shadel bits: color" << shadelColorBits
                           << " alpha" << shadelAlphaBits << " ;\\\\\\ ";
    imgInfo << endl << "compositing style: "
                           <<
TypeStr::CompositingStyle{compositingStyle} << " ; ";
25    imgInfo << endl << "cooxel bits: color" << cooxelColorBits
                           << " alpha" << cooxelAlphaBits;
    imgInfo << endl;
}
sprintf(str,"convert img/baseplaneImage.miff img/%s.eps \n",
       imageRootFileName);
system(str);
30    sprintf(str,"%s ; %s%s%s \n",
              "latex img/image.ltx > /dev/null",
              "dvips -o img/",imageRootFileName,".ps
image.dvi");
system(str);
35    sprintf(str,"ghostview img/%s.ps & \n",   imageRootFileName);
system(str);
system("/usr/bin/rm -f im*.aux ; /usr/bin/rm -f image.* ; /usr/bin/rm -f
img/*.aux");
} // MakePSfile

40 // end of Cube4Interface.C
:::::::::::
cube4/Cube4PipelineParameter.dat
:::::::::::
// dataset
45 // -nc 2 // number of cube4 chips
// -np 4 // number of pipelines
// -bs 8 // block size
// -ds data/geom/bulb.slc // dataset file

```

50

55

```

//--ir Image

5   // pipeline mode
//--pa          // projection style
//--c4          // Cube4 light mode

// Levoy classification
//--th 0.6    // threshold
10  //--mw 1.5    // maxwidth
    //--ma 10     // mulAlpha

// lights
//--sm
//--ka 0
15  //--kd 1
//--ks 0
    //--li " 0.0 0.0 1.0 1 .5 .5      1"
    //--li "-0.8 -0.5 1.0 8 8 9      60"
    //--li " 0.8 -0.5 1.0 8 8 9      60"
    //--li "-0.8 -0.5 1.0 -9 -9 0      99"
20  //--li " 0.8 -0.5 1.0 -9 -9 0      99"
    //--li " 0.0 0.4 1.0 3 0 0      50"
    //--li "-1.0 1.0 1.0 2 2 2      70"
    //--li " 1.0 1.0 1.0 2 2 2      70"
    //--li "-0.4 1.0 0.6 2 2 2      70"
    //--li " 0.4 1.0 0.6 2 2 2      70"
25  //--li " 0.0 1.0 0.6 2 2 2      99"

:::::::::::
cube4/DataSyncPipeline.C
:::::::::::
30  // DataSyncPipeline.C
    // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

35  #include "DataSyncPipeline.h"

void DataSyncPipeline::Demo()
{
    DataSyncPipeline dataSync;
    cout << endl << "Demo of class " << typeid(dataSync).name();
40  cout << endl << "size : " << sizeof(DataSyncPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "DataSyncPipeline dataSync; = " << dataSync;
    cout << endl << "End of demo of class " << typeid(dataSync).name() << endl;
} // Demo

45  /////////////////////////////////
// constructors & destructors

```

```

155
// static first init
5   int DataSyncPipeline::numOfChips          = 0;
int DataSyncPipeline::numOfPipelinesPerChip = 0;
int DataSyncPipeline::blockSize            = 0;
Cube4 *DataSyncPipeline::cube4           = 0;

DataSyncPipeline::DataSyncPipeline()
10  {
      int blockBeamDelay = blockSize / numOfPipelinesPerChip;
      int syncDelay = 2*blockBeamDelay + 3; // 2 fifo's, 3 extra stages
      blockSyncShade1FiFo.SetSize(syncDelay);
      blockSyncWeightsFiFo.SetSize(syncDelay);
      blockSyncPerPipelineControlFlagsFiFo.SetSize(syncDelay);
15  blockSyncPerChipControlFlagsFiFo.SetSize(syncDelay);
} // constructor

DataSyncPipeline::~DataSyncPipeline()
20  {
} // destructor

25  //////////////////////////////////////////////////////////////////
// show/set data & data properties

ostream & DataSyncPipeline::Ostream(ostream & os) const
{
    // append DataSyncPipeline info to os
30  os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips        = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex         = " << chipIndex;

35  // return complete os
    return os;

} // Ostream

40  //////////////////////////////////////////////////////////////////
// show/set data & data properties
// - local show/set functions

45  void DataSyncPipeline::GlobalSetup(const int setNumOfChips,
                                     const int setNumOfPipelinesPerChip,
50  const int setBlockSize)
{
    numOfChips          = setNumOfChips;
}

```

55

```

156
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize             = setBlockSize;
5   } // GlobalSetup

void DataSyncPipeline::LocalSetup(const int setChipIndex,
10
        const int setPipelineIndex,
{
        DataSyncStage & dataSyncStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

15
    inputs.shadel = &(dataSyncStage.inputs.shadel[pipelineIndex]);
    inputs.weightsXYZ = &(dataSyncStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = dataSyncStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(dataSyncStage.inputs.perPipelineControlFlags[pipelineIndex]);
20
    results.shadel = &(dataSyncStage.results.shadel[pipelineIndex]);
    results.weightsXYZ = &(dataSyncStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(dataSyncStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(dataSyncStage.results.perChipControlFlags);

    cube4 = dataSyncStage(cube4);
25
} // LocalSetup

30
void DataSyncPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in DataSyncStage
    blockSyncShadelFiFo.Preset(*(inputs.shadel));
    blockSyncWeightsFiFo.Preset(*(inputs.weightsXYZ));
35
    blockSyncPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFla
gs));
    blockSyncPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));
} // PerFrameSetup

40
///////////////////////////////
// local computation functions

void DataSyncPipeline::RunForOneClockCycle()
{
    blockSyncShadelFiFo.Exchange
45
        (*(inputs.shadel),
         *(results.shadel));
    blockSyncWeightsFiFo.Exchange
        (*(inputs.weightsXYZ),

```

50

55

```

157
    *(results.weightsXYZ));
blockSyncPerPipelineControlFlagsFiFo.Exchange
5      (*(inputs.perPipelineControlFlags),
        *(results.perPipelineControlFlags));
if (pipelineIndex == 0)
    blockSyncPerChipControlFlagsFiFo.Exchange
        (*(inputs.perChipControlFlags),
10       *(results.perChipControlFlags));
} // RunForOneClockCycle

///////////////////////////////
// internal utility functions
15

// end of DataSyncPipeline.C
:::::::::::
cube4/DataSyncPipeline.h
:::::::::::
20 // DataSyncPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.
25

#ifndef _DataSyncPipeline_h_ // prevent multiple includes
#define _DataSyncPipeline_h_

#include "Misc.h"
30 #include "Object.h"
#include "FiFo.h"
#include "Shadel.h"
#include "Control.h"
#include "FixPointNumber.h"

35 typedef Vector3D<FixPointNumber> FixPointVector3D;

class DataSyncStage;
class Cube4;

40 class DataSyncPipelineInputs {
public: // pointers
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
45    PerPipelineControlFlags *perPipelineControlFlags;
};

50 class DataSyncPipelineResults {
public: // pointers
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
};

```

55

```

158
    PerChipControlFlags           *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
5   };

class DataSyncPipeline : virtual public Object {
public:
10    static void      Demo ();

        // constructors & destructors
        DataSyncPipeline ();
        ~DataSyncPipeline ();

15    // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream & ) const;

        // - local show/set functions
20    static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           const int setBlockSize);

25    virtual void LocalSetup(const int setChipIndex,
                           const int setPipelineIndex,
                           DataSyncStage & dataSyncStage);
        virtual void PerFrameSetup();

30    // local computation functions
        virtual void RunForOneClockCycle();

public:
35    DataSyncPipelineInputs  inputs;
    DataSyncPipelineResults results;

protected:
40    Fifo<Shadel> blockSyncShadelFifo;
    Fifo<FixPointVector3D> blockSyncWeightsFifo;
    Fifo<PerPipelineControlFlags> blockSyncPerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags> blockSyncPerChipControlFlagsFifo;

        static int    numOfChips, numOfPipelinesPerChip, blockSize;
        static Cube4 *cube4;
        int          chipIndex, pipelineIndex;    // only for debugging purpose
45    };
};

#include "DataSyncStage.h"
#include "Cube4.h"

#endif      // _DataSyncPipeline_h_
50

```

159

```

:::::::::::
cube4/DataSyncStage.C
:::::::::::
5 // DataSyncStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

10 #include "DataSyncStage.h"

void DataSyncStage::Demo()
{
    DataSyncStage dataSync;
    cout << endl << "Demo of class " << typeid(dataSync).name();
    cout << endl << "size : " << sizeof(DataSyncStage).<< ' Bytes';
    cout << endl << "public member functions:";
    cout << endl << "DataSyncStage dataSync; = " << dataSync;
    cout << endl << "End of demo of class "<< typeid(dataSync).name() << endl;
} // Demo
20

///////////////////////////////
// constructors & destructors

25 // static first init
int DataSyncStage::numOfChips           = 0;
int DataSyncStage::numOfPipelinesPerChip = 0;
Cube4 *DataSyncStage::cube4 = 0;

DataSyncStage::DataSyncStage()
{
    dataSyncPipeline = new DataSyncPipeline [numOfPipelinesPerChip];
    results.shadel = new Shadel [numOfPipelinesPerChip + 1];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
    [numOfPipelinesPerChip];
} // defaultconstructor
35

DataSyncStage::~DataSyncStage()
{
    if (dataSyncPipeline) { delete dataSyncPipeline; dataSyncPipeline=0; }
    if (results.shadel)   { delete results.shadel;      results.shadel=0; }
}
40    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0; }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor
45
50

```

55

```

5   // show/set data & data properties

10  ostream & DataSyncStage::Ostream(ostream & os) const
11  {
12      // append DataSyncStage info to os
13      os << typeid(*this).name() << "@" << (void *) this;
14      os << endl << "  numOfChips          = " << numOfChips;
15      os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
16      os << endl << "  chipIndex           = " << chipIndex;

17  // return complete os
18  return os;

19 } // Ostream

20 // show/set data & data properties
21 //
22 // - local show/set functions

23 void DataSyncStage::GlobalSetup(const int setNumOfChips,
24                                 const int setNumOfPipelinesPerChip,
25                                 const int setBlockSize,
26                                 Cube4 *setCube4)
27 {
28     numOfChips          = setNumOfChips;
29     numOfPipelinesPerChip = setNumOfPipelinesPerChip;
30     DataSyncPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
31
32         setBlockSize);
33     cube4 = setCube4;
34 } // GlobalSetup

35 void DataSyncStage::LocalSetup(const int setChipIndex)
36 {
37     chipIndex = setChipIndex;
38     for (int p=0; p<numOfPipelinesPerChip; ++p) {
39         dataSyncPipeline[p].LocalSetup(chipIndex,p,*this);
40     }
41 } // LocalSetup

42 void DataSyncStage::PerFrameSetup()
43 {

```

```

161
int p;
// reset pipeline registers
5   for (p=0; p<numOfPipelinesPerChip; ++p) {
    results.shadel[p] = cube4->backgroundShadel;
    results.weightsXYZ[p](0,0,0);
    results.perPipelineControlFlags[p].Reset();
}
results.perChipControlFlags.Reset();

10  for (p=0; p<numOfPipelinesPerChip; ++p) {
    dataSyncPipeline[p].PerFrameSetup();
}

15  // print debug info
//static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

20  /////////////////
// local computation functions

void DataSyncStage::RunForOneClockCycle()
{
    // communication

25    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        dataSyncPipeline[p].RunForOneClockCycle();
    }
} // RunForOneClockCycle

30  /////////////////
// internal utility functions

35  // end of DataSyncStage.C
:::::::::::
cube4/DataSyncStage.h
:::::::::::
// DataSyncStage.h
// (c) Ingmar Bitter '97
40
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45  #ifndef _DataSyncStage_h_      // prevent multiple includes
#define _DataSyncStage_h_

#include "Misc.h"
#include "Object.h"
#include "Vector3D.h"
#include "Shadel.h"
50

```

55

162

```

5      #include "Control.h"
#include "DataSyncPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

10     class Cube4;

15     class DataSyncStageInputs {
public: // pointers
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

20     class DataSyncStageResults {
public: // arrays
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

25     class DataSyncStage : virtual public Object {
public:

30         static void Demo ();

            // constructors & destructors
        DataSyncStage ();
        ~DataSyncStage ();

35         // show/set data & data properties
            // - class Object requirements
        virtual ostream & Ostream (ostream &) const;

            // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,
                                const int setNumOfPipelinesPerChip,
                                const int setBlockSize,
                                Cube4 *setCube4);
        virtual void LocalSetup (const int setChipIndex);
        virtual void PerFrameSetup ();
        // local computation functions
        virtual void RunForOneClockCycle();

45        public:
            DataSyncPipeline *dataSyncPipeline;
};

50
55

```



```

164
int                               FinalCoxelBuffer::numOfPipelinesPerChip
= 0;
5   int FinalCoxelBuffer::blockSize           = 0;
Cube4 * FinalCoxelBuffer::cube4          = 0;

FinalCoxelBuffer::FinalCoxelBuffer()
{
10    results.coxelBurstPackage = new Coxel [blockSize];
coixelSideBuffer= new Coxel [blockSize];
coixelBuffer    = new Coxel [blockSize*blockSize];
coixelAddress   = new int   [blockSize];
} // constructor

15
FinalCoxelBuffer::~FinalCoxelBuffer()
{
    if (results.coxelBurstPackage) {
        delete results.coxelBurstPackage;
        results.coxelBurstPackage=0;
20    }
    if (coixelSideBuffer) { delete coixelSideBuffer; coixelSideBuffer=0; }
    if (coixelBuffer)     { delete coixelBuffer;     coixelBuffer=0;     }
    if (coixelAddress)   { delete coixelAddress;   coixelAddress=0;   }
} // destructor

25
///////////////////////////////
// show/set data & data properties
//
// - local show/set functions

30 ostream & FinalCoxelBuffer::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << '@' << (void *) this;

35    // return complete os
    return os;
} // Ostream

void FinalCoxelBuffer::GlobalSetup(const int setNumOfChips,
40
        const int setNumOfPipelinesPerChip,
        const int setBlockSize,
        Cube4 *setCube4)
45
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
}

50

```

```

      165
    cube4          = setCube4;
} // GlobalSetup

5

void FinalCoxelBuffer::LocalSetup(CoxMem * memory)
{
    mem = memory;
    mem->SetSize(3 * blockSize*blockSize);
10   currentBufferReadRow(0,blockSize);
    currentBufferWriteRow(0,blockSize);
    clockCyclesBeforeNextDMA = 0;
} // LocalSetup

15 void FinalCoxelBuffer::PerFrameSetup()
{
    nextBufferOffset = 0;
    datasetSizeXYZ = cube4->datasetSizeXYZ;

20   lineExtend = datasetSizeXYZ.X() / numOfChips;
    sliceExtend = lineExtend * datasetSizeXYZ.Y();

    currentBackPos = 0;
    currentTopBottomPos = sliceExtend;
25   currentSidePos = 2*sliceExtend;

    // reset pipeline registers
    results.address = 0;
    results.packageReady = false;
    for (int k=0; k<blockSize; ++k) {
        results.coxelBurstPackage[k] = 0;
    }
    // print debug info
    // static bool first(true);    if (first) { cout<<this<<endl; first =
false; }
35   } // PerFrameSetup

///////////////////////////////
// local computation functions

40 void FinalCoxelBuffer::RunForOneClockCycle()
{
    /*
45      p=0 p=1 p=2 ...
      +---+---+---+---+---+
      |   |   |   |   |   |   | currentBufferRead/WriteRow = 0
      +---+---+---+---+---+
      |   |   |   |   |   |   | currentBufferRead/WriteRow = 1
      +---+---+---+---+---+
50      |   |   |   |   |   |   | currentBufferRead/WriteRow = 2

```

```

166
+---+---+---+---+---+---+---+
| | | | | | | | ...
+---+---+---+---+---+---+
| | | | | | | | X = (p +
+---+---+---+---+---+---+
| | | | | | | |
+---+---+---+---+---+
*/
15 assert(blockSize == numOfPipelinesPerChip);

// handle back face coxels
if (inputs.perChipControlFlags[0].endFace & BackFace) {
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        coxelBuffer[p + int(currentBufferWriteRow)*blockSize]
20            = inputs.coxel[p];
    }
    coxelAddress[currentBufferWriteRow] = currentBackPos;
    ++currentBufferWriteRow;
    currentBackPos += blockSize;
}
25 else {

    // handle top/bottom face coxels
    if (inputs.perChipControlFlags[0].endFace & TopBottomFace) {
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            coxelBuffer[p + int(currentBufferWriteRow)*blockSize]
30            = inputs.coxel[p];
        }
        coxelAddress[currentBufferWriteRow] = currentTopBottomPos;
        ++currentBufferWriteRow;
        currentTopBottomPos += blockSize;
    }
35

    // handle side face coxels
    if (inputs.perChipControlFlags[0].endFace & SideFace) {
        // left side
        if (inputs.perPipelineControlFlags[0].endOfRay) {
40            coxelSideBuffer[nextBufferOffset] = inputs.coxel[0];
        }
        // right side
        else {

            assert(inputs.perPipelineControlFlags[numOfPipelinesPerChip-1].endOfRay ==
45            true);
            coxelSideBuffer[nextBufferOffset]
                = inputs.coxel[numOfPipelinesPerChip-1];
        }
    }
}
50

```

55

```

5          // both sides
      ++nextBufferOffset;
      if (nextBufferOffset == blockSize) {
          // copy coxelSideBuffer to coxelBuffer
          for (int p=0; p<numOfPipelinesPerChip; ++p) {
              coxelBuffer[p +
20            int(currentBufferWriteRow)*blockSize]
                  = coxelSideBuffer[p];
              coxelSideBuffer[p] = Coxel(0,1,0,1);
          }
          coxelAddress[currentBufferWriteRow] = currentSidePos;

15          // update counters
          nextBufferOffset = 0;
          currentSidePos += blockSize;
          ++currentBufferWriteRow;
      }
20    }

25    /////////////////////////////////
// assemble coxelBurstPackage

30    // wait for previous DMA transfer to finish
    if (clockCyclesBeforeNextDMA > 0) {
        --clockCyclesBeforeNextDMA;
        results.packageReady = false;
    }

35    // if previous DMA transfer is finished
    else {
        if (currentBufferReadRow == currentBufferWriteRow) {
            // no row complete for DMA
            results.packageReady = false;
        }
        else {
            // buffer has at least one row ready to go
            for (int p=0; p<numOfPipelinesPerChip; ++p) {
40            results.coxelBurstPackage[p] =
                coxelBuffer[p +
                    int(currentBufferReadRow)*blockSize];
            }
            results.address = coxelAddress[int(currentBufferReadRow)];
            results.packageReady = true;
        }
45        ++currentBufferReadRow;
        clockCyclesBeforeNextDMA = 1; // blockSize;
    }
50}

```

```

////////// // send coxelBurstPackage
5   if (results.packageReady == true) {
        for (int p=0; p<numOfPipelinesPerChip; ++p) {
            (*mem)(results.address+p) = results.coxelBurstPackage[p];
        }
10
        //cout <<"burst to ["<<results.address<<"]@clk"<<cube4->clk<< " ;
        if (results.address == datasetSizeXYZ.X()*(datasetSizeXYZ.Y()-1)) {
            // final back face row was written, work done
            cout << "last Coxel package written to SDRAM at cc "<<cube4-
>clk<<endl;
15
        cube4->clk = -10;
    }
}
} // RunForOneClockCycle

20 // end of FinalCoxelBuffer.C
:::::::::::
cube4/FinalCoxelBuffer.h
:::::::::::
// FinalCoxelBuffer.h
// (c) Ingmar Bitter '97
25

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// FinalCoxelBuffer
30 // in : final pixels from registers after compos unit in unregular amounts
// out: pixels in DMA burst mode packages

#ifndef _FinalCoxelBuffer_h_ // prevent multiple includes
#define _FinalCoxelBuffer_h_

35 #include "Object.h"
#include "Control.h"
#include "Coxel.h"
#include "CoxMem.h"
#include "ModInt.h"
#include "Vector3D.h"
40

40 class Cube4;

class FinalCoxelBufferInputs {
public: // pointers
45     Coxel *coixel;
    PerPipelineControlFlags *perPipelineControlFlags;
    PerChipControlFlags     *perChipControlFlags;
};

50

```

```

5      class FinalCoxelBufferResults {
public: // arrays
       Coxel *coixelBurstPackage;
       int   address;
       bool  packageReady;
};

10
11      class FinalCoxelBuffer : virtual public Object {
public:

15          static void      Demo ();
16
17          // constructors & destructors
18          FinalCoxelBuffer();
19          ~FinalCoxelBuffer();
20
21          // show/set data & data properties
22          virtual         ostream &   Ostream (ostream & ) const;

23          static  void GlobalSetup (const int setNumOfChips,
24
25          const int setNumOfPipelinesPerChip,
26
27          const int setBlockSize,
28
29          Cube4 *setCube4);
30          virtual void LocalSetup(CoxMem * memory);
31          virtual void PerFrameSetup();

32
33          // local computation functions
34          virtual void RunForOneClockCycle();
35          //static int BackArrayAdrr();
36          //static int BottomArrayAdrr();
37          //static int SideArrayAdrr();

38
39          public:
40              FinalCoxelBufferInputs inputs;
41              FinalCoxelBufferResults results;
42
43          protected:
44              Coxel *coixelSideBuffer; // b buffer
45              Coxel *coixelBuffer;    // b^2 buffer
46              int   *coixelAddress;   // b buffer
47              int nextBufferOffset;
48              ModInt currentBufferReadRow;
49              ModInt currentBufferWriteRow;
50              static int numOfChips, numOfPipelinesPerChip, blockSize;
51              Vector3D<int> datasetSizeXYZ;
52              int lineExtend; // this many addresses are needed per line
53              int sliceExtend; // this many addresses are needed per slice
54              int currentBackPos;
55              int currentTopBottomPos;

```

```

170
    int currentSidePos;
    int clockCyclesBeforeNextDMA;

5     CoxMem *mem;
     static Cube4 *cube4;

     friend class Cube4;
};

10    #include "Cube4.h"

#endif      // _FinalCoxelBuffer_h_
:::::::::::
cube4/GradXPipeline.C
:::::::::::
// GradXPipeline.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
20   // America, Inc., 1997, All rights reserved.

#include "GradXPipeline.h"

void GradXPipeline::Demo()
{
25     GradXPipeline gradX;
     cout << endl << "Demo of class " << typeid(gradX).name();
     cout << endl << "size : " << sizeof(GradXPipeline) << " Bytes";
     cout << endl << "public member functions:";
     cout << endl << "GradXPipeline gradX; = " << gradX;
     cout << endl << "End of demo of class " << typeid(gradX).name() << endl;
} // Demo

///////////////////////////////
// constructors & destructors
35

// static first init
int GradXPipeline::numOfChips          = 0;
int GradXPipeline::numOfPipelinesPerChip = 0;
Cube4 *GradXPipeline::cube4            = 0;

40

GradXPipeline::GradXPipeline()
{
} // constructor

45

GradXPipeline::~GradXPipeline()
{
} // destructor

```

50

55

```

5      //////////////////////////////////////////////////////////////////
// show/set data & data properties

10     ostream & GradXPipeline::Ostream(ostream & os) const
{
    // append GradXPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;

15     // return complete os
    return os;

} // Ostream

20     //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

25     void GradXPipeline::GlobalSetup(const int setNumOfChips,
{
    const int setNumOfPipelinesPerChip)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
} // GlobalSetup

35     void GradXPipeline::LocalSetup(const int setChipIndex,
{
    const int setPipelineIndex,
    GradXStage & gradXStage)
{
    chipIndex = setChipIndex;
    pipelineIndex(setPipelineIndex, numOfPipelinesPerChip+2); // ModInt init
40
    inputs.voxelXa = &(gradXStage.inputs.voxel[pipelineIndex-1]);
    inputs.voxelXb = &(gradXStage.inputs.voxel[pipelineIndex+1]);
    inputs.weightsXYZ = &(gradXStage.inputs.weightsXYZ[int(pipelineIndex)]);
    inputs.perChipControlFlags
        = gradXStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(gradXStage.inputs.perPipelineControlFlags[pipelineIndex]);

45
    results.voxel = &(gradXStage.results.voxel[pipelineIndex]);
    results.gx = &(gradXStage.results.gx[pipelineIndex]);
} // LocalSetup

```

50

55

```

172
    results.gy =
&(gradXStage.results.gy[pipelineIndex]);
5     results.weightsXYZ = &(gradXStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(gradXStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(gradXStage.results.perChipControlFlags);

10    cube4 = gradXStage(cube4;
} // LocalSetup

void GradXPipeline::PerFrameSetup()
15 {
    // reset pipeline registers already done in GradXStage
} // PerFrameSetup

20 /////////////////
// local computation functions

void GradXPipeline::RunForOneClockCycle()
{
25     *(results.gx) =
        (ScalarGradient) inputs.voxelXb->raw16bit -
        (ScalarGradient) inputs.voxelXA->raw16bit;

} // RunForOneClockCycle

30 /////////////////
// internal utility functions

35 // end of GradXPipeline.C
:::::::::::
cube4/GradXPipeline.h
:::::::::::
// GradXPipeline.h
// (c) Ingmar Bitter '97
40
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45 #ifndef _GradXPipeline_h_      // prevent multiple includes
#define _GradXPipeline_h_

50 #include "Misc.h"
#include "Object.h"
#include "ModInt.h"
#include "FiFo.h"
#include "Voxel.h"
#include "Coxel.h"

```

```

5      #include "Control.h"
6      #include "FixPointNumber.h"

7      class GradXStage;
8      class Cube4;

9      class GradXPipelineInputs {
10         public: // pointers
11             Voxel *voxelXA;
12             Voxel *voxelXB;
13             Vector3D<FixPointNumber> *weightsXYZ;
14             PerChipControlFlags    *perChipControlFlags;
15             PerPipelineControlFlags *perPipelineControlFlags;
16         };

17
18
19
20     class GradXPipelineResults {
21         public: // pointers
22             Voxel *voxel;
23             ScalarGradient *gx, *gy;
24             Vector3D<FixPointNumber> *weightsXYZ;
25             PerChipControlFlags    *perChipControlFlags;
26             PerPipelineControlFlags *perPipelineControlFlags;
27         };

28
29
30     class GradXPipeline : virtual public Object {
31         public:
32             static void      Demo ();
33
34             // constructors & destructors
35             GradXPipeline ();
36             ~GradXPipeline ();
37
38             // show/set data & data properties
39             // - class Object requirements
40             virtual ostream & Ostream (ostream & ) const;
41
42             // - local show/set functions
43             static void GlobalSetup (const int setNumOfChips,
44
45             const int setNumOfPipelinesPerChip);
46
47             virtual void LocalSetup(const int setChipIndex,
48
49             const int setPipelineIndex,
50
51             GradXStage & gradXStage);
52             virtual void PerFrameSetup();
53             // local computation functions
54             virtual void RunForOneClockCycle();
55

```

```

public:
    GradXPipelineInputs inputs;
    GradXPipelineResults results;

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    int          chipIndex;
    ModInt      pipelineIndex;
};

#include "GradXStage.h"
#include "Cube4.h"

#endif // _GradXPipeline_h_
::::::::::
cube4/GradXStage.C
::::::::::
// GradXStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

#include "GradXStage.h"
void GradXStage::Demo()
{
    GradXStage gradX;
    cout << endl << "Demo of class " << typeid(gradX).name();
    cout << endl << "size : " << sizeof(GradXStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "GradXStage gradX; = " << gradX;
    cout << endl << "End of demo of class " << typeid(gradX).name() << endl;
} // Demo

///////////////////////////////
// constructors & destructors

// static first init
int GradXStage::numOfChips           = 0;
int GradXStage::numOfPipelinesPerChip = 0;
Cube4 *GradXStage::cube4 = 0;

GradXStage::GradXStage()
{
    gradXPipeline = new GradXPipeline [numOfPipelinesPerChip];
    results.voxel = new Voxel [numOfPipelinesPerChip];
    results.gx = new ScalarGradient [numOfPipelinesPerChip];
    results gy = new ScalarGradient [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
}

```

50

55

```

      175
    results.perPipelineControlFlags = new PerPipelineControlFlags
[ numOfPipelinesPerChip];
} // defaultconstructor

5

GradXStage::~GradXStage()
{
    if (gradXPipeline) { delete gradXPipeline; gradXPipeline=0; }
    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.gx) { delete results.gx; results.gx=0; }
    if (results gy) { delete results.gy; results.gy=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0; }

10   if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

15

20 /////////////////////////////////
// show/set data & data properties

25 ostream & GradXStage::Ostream(ostream & os) const
{
    // append GradXStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "numOfChips"           = " << numOfChips;
    os << endl << "numOfPipelinesPerChip" = " << numOfPipelinesPerChip;
    os << endl << "chipIndex"           = " << chipIndex;

30   // return complete os
    return os;

} // Ostream

35

40 /////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

45 void GradXStage::GlobalSetup(const int setNumOfChips,
                               const int setNumOfPipelinesPerChip,
                               Cube4 *setCube4)
{
    numOfChips           = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    GradXPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip);

50
}

```

55

176

```

    cube4 = setCube4;
} // GlobalSetup
5

void GradXStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradXPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup

15 void GradXStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.gx[p] = results gy[p] = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

20    for (p=0; p<numOfPipelinesPerChip; ++p) {
        gradXPipeline[p].PerFrameSetup();
    }

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
30 } // PerFrameSetup

///////////////////////////////
// local computation functions
35 void GradXStage::RunForOneClockCycle()
{
    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradXPipeline[p].RunForOneClockCycle();
        results.voxel[p] = inputs.voxel[p];
        results.gy[p] = inputs.gy[p];
        results.weightsXYZ[p] = inputs.weightsXYZ[p];
        results.perPipelineControlFlags[p] =
inputs.perPipelineControlFlags[p];
    }
    results.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle

50 ///////////////////////

```

55

```

// internal utility functions

5   // end of GradXStage.C
:::::::::::
cube4/GradXStage.h
:::::::::::
// GradXStage.h
10  // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15  #ifndef _GradXStage_h_ // prevent multiple includes
#define _GradXStage_h_

20  #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Control.h"
#include "GradXPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

25  class Cube4;

class GradXStageInputs {
public: // pointers
30  Voxel *voxel;
ScalarGradient *gy;
Vector3D<FixPointNumber> *weightsXYZ;
PerChipControlFlags *perChipControlFlags;
PerPipelineControlFlags *perPipelineControlFlags;
};

35

class GradXStageResults {
public: // arrays
Voxel *voxel;
ScalarGradient *gx, *gy;
Vector3D<FixPointNumber> *weightsXYZ;
PerChipControlFlags perChipControlFlags;
PerPipelineControlFlags *perPipelineControlFlags;
};

40

45  class GradXStage : virtual public Object {
public:

50  static void Demo ();
// constructors & destructors
GradXStage ();

```

```

178
~GradXStage ();

5    // show/set data & data properties
    // - class Object requirements
    virtual ostream & Ostream (ostream &) const;

    // - local show/set functions
10   static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           Cube4 *setCube4);
    virtual void LocalSetup (const int setChipIndex);
    virtual void PerFrameSetup ();
15   // local computation functions
    virtual void RunForOneClockCycle();

public:
20   GradXPipeline *gradXPipeline;
    GradXStageInputs inputs;
    GradXStageResults results;

protected:
25   static int numOfChips, numOfPipelinesPerChip;
   static Cube4 *cube4;
   int chipIndex;

   friend class GradXPipeline;
};

30 #endif // _GradXStage_h_
::::::::::
cube4/GradYPipeline.C
::::::::::
// GradYPipeline.C
// (c) Ingmar Bitter '97
35

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "GradYPipeline.h"
40
void GradYPipeline::Demo()
{
    GradYPipeline gradY;
    cout << endl << "Demo of class " << typeid(gradY).name();
    cout << endl << "size : " << sizeof(GradYPipeline) << " Bytes";
45   cout << endl << "public member functions:";
    cout << endl << "GradXStage gradY; = " << gradY;
    cout << endl << "End of demo of class " << typeid(gradY).name() << endl;
} // Demo

```

50

```

////////// // constructors & destructors

5   // static first init
int GradYPipeline::numOfChips          = 0;
int GradYPipeline::numOfPipelinesPerChip = 0;
int GradYPipeline::blockSize           = 0;
10  Cube4 *GradYPipeline::cube4         = 0;

GradYPipeline::GradYPipeline()
{
    int partialBeamsPerBlockBeam = blockSize / numOfPipelinesPerChip;

15  blockBeamVoxelFiFo0.SetSize(partialBeamsPerBlockBeam);
    blockBeamVoxelFiFo1.SetSize(partialBeamsPerBlockBeam);
    blockBeamWeightsFiFo.SetSize(partialBeamsPerBlockBeam);
    blockBeamPerChipControlFlagsFiFo.SetSize(partialBeamsPerBlockBeam);
    blockBeamPerPipelineControlFlagsFiFo.SetSize(partialBeamsPerBlockBeam);
20  } // constructor

GradYPipeline::~GradYPipeline()
{
} // destructor

////////// // show/set data & data properties

25

30  ostream & GradYPipeline::Ostream(ostream & os) const
{
    // append GradYPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "numOfChips"           = " " << numOfChips;
35  os << endl << "numOfPipelinesPerChip" = " " << numOfPipelinesPerChip;
    os << endl << "chipIndex"           = " " << chipIndex;

    // return complete os
    return os;
40  } // Ostream

////////// // show/set data & data properties
45  //
// - local show/set functions

50  void GradYPipeline::GlobalSetup(const int setNumOfChips,

```

180

```

    const int setNumOfPipelinesPerChip,
5           const int setBlockSize)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
10      } // GlobalSetup

void GradYPipeline::LocalSetup(const int setChipIndex,
                               const int setPipelineIndex,
15      GradYStage & gradYStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

20      inputs.voxel = &(gradYStage.inputs.voxel[pipelineIndex]);
    inputs.weightsXYZ = &(gradYStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = gradYStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(gradYStage.inputs.perPipelineControlFlags[pipelineIndex]);
25      results.voxel = &(gradYStage.results.voxel[pipelineIndex]);
    results.gy = &(gradYStage.results.gy[pipelineIndex]);
    results.weightsXYZ = &(gradYStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(gradYStage.results.perPipelineControlFlags[pipelineIndex]);
30      results.perChipControlFlags
        = &(gradYStage.results.perChipControlFlags);

    cube4 = gradYStage(cube4);
} // LocalSetup

35 void GradYPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in GradYStage
    blockBeamVoxelFiFo0.Preset(*inputs.voxel);
    blockBeamVoxelFiFo1.Preset(*inputs.voxel);
40    blockBeamWeightsFiFo.Preset(*inputs.weightsXYZ);
    blockBeamPerPipelineControlFlagsFiFo.Preset(*inputs.perPipelineControlFla
gs));
    blockBeamPerChipControlFlagsFiFo.Preset(*inputs.perChipControlFlags);
} // PerFrameSetup

45 //////////////////////////////////////////////////////////////////
// local computation functions

```

50

55

```

void GradYPipeline::RunForOneClockCycle()
{
    /*
        ----->(x)
        |
        | voxA (seen early during processing => take from Fifo )
        |
        | voxB (seen late during processing => take from input)
        |
        V(y)
    */

    15   blockBeamVoxelFifo0.Exchange( *(inputs.voxel), current );
    blockBeamVoxelFifo1.Exchange( current, lastFifoOut );
    blockBeamWeightsFifo.Exchange( *(inputs.weightsXYZ),
                                    *(results.weightsXYZ));
    20   blockBeamPerPipelineControlFlagsFifo.Exchange
          (*(inputs.perPipelineControlFlags),
           *(results.perPipelineControlFlags));
    if (pipelineIndex == 0)
        blockBeamPerChipControlFlagsFifo.Exchange
          (*(inputs.perChipControlFlags),
           *(results.perChipControlFlags));

    25   *(results.gy) = // voxB - voxA
          (ScalarGradient) inputs.voxel->raw16bit -
          (ScalarGradient) lastFifoOut.raw16bit;

    30   *(results.voxel) = current;
} // RunForOneClockCycle

35   /////////////////////////////////
// internal utility functions

// end of GradYPipeline.C
:::::::::::
40   cube4/GradYPipeline.h
:::::::::::
// GradYPipeline.h
// (c) Ingmar Bitter '97

45   // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _GradYPipeline_h_      // prevent multiple includes
#define _GradYPipeline_h_

50   #include "Misc.h"
#include "Object.h"

```

182

```

5      #include "FIFO.h"
# include "Voxel.h"
# include "Coxel.h"
# include "Control.h"
# include "FixPointNumber.h"

10     class GradYStage;
class Cube4;

15     typedef Vector3D<FixPointNumber> FixPointVector3D;

class GradYPipelineInputs {
public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

20

class GradYPipelineResults {
public: // pointers
    Voxel *voxel;
    ScalarGradient *gy;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

25

30     class GradYPipeline : virtual public Object {
public:

35         static void Demo ();

        // constructors & destructors
        GradYPipeline ();
        ~GradYPipeline ();

40         // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream & ) const;

        // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,
50            const int setNumOfPipelinesPerChip,
            const int setBlockSize);

        virtual void LocalSetup(const int setChipIndex,
            const int setPipelineIndex,

```

55

```

5           GradYStage & gradYStage);
virtual void PerFrameSetup();
// local computation functions
virtual void RunForOneClockCycle();

10      public:
GradYPipelineInputs inputs;
GradYPipelineResults results;

protected:
FIFO<Voxel> beamVoxelFiFo0;
FIFO<Voxel> beamVoxelFiFo1;
15      FIFO<FixPointVector3D> beamWeightsFiFo;
FIFO<PerPipelineControlFlags> beamPerPipelineControlFlagsFiFo;
FIFO<PerChipControlFlags> beamPerChipControlFlagsFiFo;

20      FIFO<Voxel> blockBeamVoxelFiFo0;
FIFO<Voxel> blockBeamVoxelFiFo1;
FIFO<FixPointVector3D> blockBeamWeightsFiFo;
FIFO<PerPipelineControlFlags> blockBeamPerPipelineControlFlagsFiFo;
FIFO<PerChipControlFlags> blockBeamPerChipControlFlagsFiFo;

25      Voxel current, lastFiFoOut;

static int numOfcHips, numOfpipelinesPerChip, blockSize;
static Cube4 *cube4;
int chipIndex, pipelineIndex;

30      friend class Cube4;
};

# include "GradYStage.h"
# include "Cube4.h"

35      #endif // _GradYPipeline_h_
:::::::::::
cube4/GradYStage.C
:::::::::::
40      // GradYStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45      #include "GradYStage.h"

void GradYStage::Demo()
{
    GradYStage gradY;
    cout << endl << "Demo of class " << typeid(gradY).name();
    cout << endl << "size : " << sizeof(GradYStage) << " Bytes";
    cout << endl << "public member functions:";

50

```

```

184
    cout << endl << "GradYStage gradY;      = " << gradY;
    cout << endl << "End of demo of class " << typeid(gradY).name() << endl;
} // Demo
5

///////////////////////////////
// constructors & destructors

10   // static first init
int GradYStage::numOfChips          = 0;
int GradYStage::numOfPipelinesPerChip = 0;
Cube4 *GradYStage::cube4 = 0;

15   GradYStage::GradYStage()
{
    gradYPipeline = new GradYPipeline [numOfPipelinesPerChip];
    results.voxel = new Voxel [numOfPipelinesPerChip+2];
    results.gy = new ScalarGradient [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
20   [numOfPipelinesPerChip];
} // defaultconstructor

25   GradYStage::~GradYStage()
{
    if (gradYPipeline) { delete gradYPipeline; gradYPipeline=0; }
    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.gy) { delete results.gy; results.gy=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0; }
}
30   if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

35   /////////////////////////
// show/set data & data properties

40   ostream & GradYStage::Ostream(ostream & os) const
{
    // append GradYStage info to os
    os << typeid(*this).name() << "@ " << (void *) this;
    os << endl << "  numOfChips           = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;
45   // return complete os
    return os;
}

50

```

```

} // Ostream

5   /////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

10 void GradYStage::GlobalSetup(const int setNumOfChips,
                               const int setNumOfPipelinesPerChip,
                               const int setBlockSize,
15   Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    GradYPipeline::GlobalSetup(numOfChips,numOfPipelinesPerChip, setBlockSize);
20   cube4 = setCube4;
} // GlobalSetup

void GradYStage::LocalSetup(const int setChipIndex)
25 {
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradYPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup
30

void GradYStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
35   for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.gy[p] = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
40   results.perChipControlFlags.Reset();

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        gradYPipeline[p].PerFrameSetup();
    }
45   // print debug info
   //static bool first(true); if (first) { cout<<this<<endl; first=false; }

} // PerFrameSetup
50

```

```

////////// local computation functions

5   // local computation functions

void GradYStage::RunForOneClockCycle()
{
    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradYPipeline[p].RunForOneClockCycle();
    }
} // RunForOneClockCycle

15  ////////// internal utility functions

// end of GradYStage.C
20  ::::::::::::::
cube4/GradYStage.h
::::::::::::
// GradYStage.h
// (c) Ingmar Bitter '97

25  // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _GradYStage_h_ // prevent multiple includes
#define _GradYStage_h_
30
#include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Control.h"
35
#include "GradYPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

class Cube4;

40  class GradYStageInputs {
public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
45
    PerPipelineControlFlags *perPipelineControlFlags;
};

50  class GradYStageResults {
public: // arrays
    Voxel *voxel;
    ScalarGradient *gy;
}

```

```

188
// America, Inc., 1997, All rights reserved.

5 #include "GradZLinXPipeline.h"

void GradZLinXPipeline::Demo()
{
    GradZLinXPipeline gradZLinX;
    cout << endl << "Demo of class " << typeid(gradZLinX).name();
    cout << endl << "size : " << sizeof(GradZLinXPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "GradZLinXPipeline gradZLinX; = " << gradZLinX;
    cout << endl << "End of demo of class " << typeid(gradZLinX).name() <<
    endl;
} // Demo
15

///////////////////////////////
// constructors & destructors

20 // static first init
int GradZLinXPipeline::numOfChips          = 0;
int GradZLinXPipeline::numOfPipelinesPerChip = 0;
int GradZLinXPipeline::blockSize           = 0;
Cube4 *GradZLinXPipeline::cube4            = 0;

25 GradZLinXPipeline::GradZLinXPipeline()
{
    int blockBeamDelay = blockSize / numOfPipelinesPerChip;
    blockBeamGradientFiFo.SetSize(blockBeamDelay);
30 } // constructor

GradZLinXPipeline::~GradZLinXPipeline()
{
} // destructor
35

///////////////////////////////
// show/set data & data properties

40 ostream & GradZLinXPipeline::Ostream(ostream & os) const
{
    // append GradZLinXPipeline info to os
    os << typeid(*this).name() << "@<" << (void *) this;
    os << endl << "    numOfChips           = " << numOfChips;
45    os << endl << "    numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "    chipIndex             = " << chipIndex;

    // return complete os
    return os;
50

```

55

189

```

} // Ostream

5   //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

10 void GradZLinXPipeline::GlobalSetup(const int setNumOfChips,
                                     const int setNumOfPipelinesPerChip,
                                     const int setBlockSize)
15 {
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize           = setBlockSize;
} // GlobalSetup

20 void GradZLinXPipeline::LocalSetup(const int setChipIndex,
                                     const int setPipelineIndex,
                                     GradZLinXStage & gradZLinXStage)
25 {
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.gz = &(gradZLinXStage.inputs.gz[pipelineIndex]);
    rightNeighborGz = &(gradZLinXStage.inputs.gz[pipelineIndex+1]);
    inputs.weightsXYZ = &(gradZLinXStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = gradZLinXStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(gradZLinXStage.inputs.perPipelineControlFlags[pipelineIndex]);

30    tempResults.gz = &(gradZLinXStage.tempResults.gz[pipelineIndex]);
    tempResults.perChipControlFlags
        = &(gradZLinXStage.tempResults.perChipControlFlags);
    results.gz = &(gradZLinXStage.results.gz[pipelineIndex]);
    results.perChipControlFlags
        = &(gradZLinXStage.results.perChipControlFlags);

35    cube4 = gradZLinXStage(cube4);
} // LocalSetup

40 void GradZLinXPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in GradZLinXStage
    blockBeamGradientFiFo.Preset(*(inputs.gz));
}

45
50
55

```

190

```

} // PerFrameSetup

5   /////////////////////////////////
// local computation functions

void GradZLinXPipeline::RunForOneClockCycle()
{
    blockBeamGradientFiFo.Exchange(*(tempResults.gz),*(results.gz));

    // Linear interpolation
    // c = w(b-a)+a
    //
15   // w=0 -> c=a
    // w=1 -> c=b

    if (cube4->cubeMode == Cube4Light) {
        // just pick current voxel (as if w=0)
        *(tempResults.gz) = *(inputs.gz); // w=0;
20      // use weights later on in composing
    }

    else if (cube4->cubeMode == Cube4Classic) {
        // do real interpolation
        FixPointNumber a((int)*(inputs.gz)),
25          b((int)*(rightNeighborGz));

        *(tempResults.gz) = (ScalarGradient). (inputs.weightsXYZ->X() * (b - a) +
a);
        }
30 } // RunForOneClockCycle

/////////////////////////////
// internal utility functions

35 // end of GradZLinXPipeline.C
:::::::::::
cube4/GradZLinXPipeline.h
:::::::::::
40 // GradZLinXPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

45 #ifndef _GradZLinXPipeline_h_ // prevent multiple includes
#define _GradZLinXPipeline_h_

#include "Misc.h"
#include "Object.h"
#include "FiFo.h"
50

```

55

191

```

5      #include "Voxel.h"
      #include "Coxel.h"
      #include "Control.h"
      #include "FixPointNumber.h"

10     class GradZLinXStage;
      class Cube4;

15     class GradZLinXPipelineInputs {
public: // pointers
      ScalarGradient *gz; // partially interpolated
      Vector3D<FixPointNumber> *weightsXYZ;
      PerChipControlFlags *perChipControlFlags;
      PerPipelineControlFlags *perPipelineControlFlags;
    };

20     class GradZLinXPipelineResults {
public: // pointers
      ScalarGradient *gz; // now also x-interpolated
      PerChipControlFlags *perChipControlFlags;
    };

25     class GradZLinXPipeline : virtual public Object {
public:

30       static void Demo ();

35       // constructors & destructors
       GradZLinXPipeline ();
       ~GradZLinXPipeline ();

40       // show/set data & data properties
       // - class Object requirements
       virtual ostream & Ostream (ostream &) const;

       // - local show/set functions
       static void GlobalSetup (const int setNumOfChips,
                               const int setNumOfPipelinesPerChip,
                               const int setBlockSize);

45       virtual void LocalSetup (const int setChipIndex,
                               const int setPipelineIndex,
                               GradZLinXStage & gradZLinXStage);
       virtual void PerFrameSetup ();
// local computation functions
       virtual void RunForOneClockCycle();
};

55

```

55

192

```

public:
    GradZLinXPipelineInputs inputs;
    GradZLinXPipelineResults tempResults;
    GradZLinXPipelineResults results;

protected:
    ScalarGradient *rightNeighborGz;
    Fifo<ScalarGradient> blockBeamGradientFifo;

    static int    numOfChips, numOfPipelinesPerChip, blockSize;
    static Cube4 *cube4;
    int          chipIndex, pipelineIndex;
};

#include "GradZLinXStage.h"
#include "Cube4.h"

#endif      // _GradZLinXPipeline_h_
:::::::::::
cube4/GradZLinXStage.C
:::::::::::
// GradZLinXStage.C
// (c) Ingmar Bitter '97
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "GradZLinXStage.h"

void GradZLinXStage::Demo()
{
    GradZLinXStage gradZLinX;
    cout << endl << "Demo of class " << typeid(gradZLinX).name();
    cout << endl << "size : " << sizeof(GradZLinXStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "GradZLinXStage gradZLinX; = " << gradZLinX;
    cout << endl << "End of demo of class " << typeid(gradZLinX).name() << endl;
} // Demo

///////////////////////////////
// constructors & destructors

// static first init
int GradZLinXStage::numOfChips        = 0;
int GradZLinXStage::numOfPipelinesPerChip = 0;
int GradZLinXStage::blockSize         = 0;
Cube4 *GradZLinXStage::cube4 = 0;

GradZLinXStage::GradZLinXStage()
{

```

55

```

193
gradZLinXPipeline = new           GradZLinXPipeline
[numOfPipelinesPerChip];
5      tempResults.gz = new ScalarGradient [numOfPipelinesPerChip];
results.gz = new ScalarGradient [numOfPipelinesPerChip];

int blockBeamDelay = blockSize / numOfPipelinesPerChip;
blockBeamControlFiFo.SetSize(blockBeamDelay);
} // constructor
10

GradZLinXStage::~GradZLinXStage()
{
    if (gradZLinXPipeline) { delete gradZLinXPipeline; gradZLinXPipeline=0;
}
15    if (tempResults.gz)     { delete tempResults.gz;     tempResults.gz=0; }
    if (results.gz)        { delete results.gz;        results.gz=0; }
} // destructor

20
////////////////////////////// show/set data & data properties

ostream & GradZLinXStage::Ostream(ostream & os) const
{
25    // append GradZLinXStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "numOfChips"          = " " << numOfChips;
    os << endl << "numOfPipelinesPerChip" = " " << numOfPipelinesPerChip;
    os << endl << "chipIndex"          = " " << chipIndex;

30    // return complete os
    return os;
}

35
////////////////////////////// show/set data & data properties
//
// - local show/set functions

40 void GradZLinXStage::GlobalSetup(const int setNumOfChips,
                                   
                                    const int setNumOfPipelinesPerChip,
                                   
                                    const int setBlockSize,
                                   
45        Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
}
50

```

```

194
    blockSize          = setBlockSize;
    cube4             = setCube4;
5     GradZLinXPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
    blockSize);
) // GlobalSetup

10   void GradZLinXStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinXPipeline[p].LocalSetup(chipIndex,p,*this);
    }
15 } // LocalSetup

void GradZLinXStage::PerFrameSetup()
{
20   int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.gz[p] = 0;
    }
    tempResults.perChipControlFlags.Reset();
    results.perChipControlFlags.Reset();
25   blockBeamControlFiFo.Preset(tempResults.perChipControlFlags);

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinXPipeline[p].PerFrameSetup();
    }

30   // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

35   /////////////////////////////////
// local computation functions

void GradZLinXStage::RunForOneClockCycle()
{
40   // communication
    inputs.gz[numOfPipelinesPerChip] = 0;
    //           = cube4->gradZLinX[chipIndex+1].inputs.voxel[0];

    // computation
    blockBeamControlFiFo.Exchange(tempResults.perChipControlFlags,
45       results.perChipControlFlags);

    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinXPipeline[p].RunForOneClockCycle();
    }
50

```

55

```

      tempResults.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle
5

///////////////////////////////
// internal utility functions

10 // end of GradZLinXStage.C
:::::::::::
cube4/GradZLinXStage.h
:::::::::::
// GradZLinXStage.h
15 // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

20 #ifndef _GradZLinXStage_h_    // prevent multiple includes
#define _GradZLinXStage_h_

25 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Control.h"
#include "GradZLinXPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

30 class Cube4;

35 class GradZLinXStageInputs {
public: // pointers
    ScalarGradient *gz; // partially interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags     *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

40 class GradZLinXStageResults {
public: // arrays
    ScalarGradient *gz; // now also x-interpolated
    PerChipControlFlags     perChipControlFlags;
};

45

50 class GradZLinXStage : virtual public Object {
public:
    static void      Demo ();
}

// constructors & destructors

```

196

```

5           GradZLinXStage ();
6           ~GradZLinXStage ();

10          // show/set data & data properties
11          // - class Object requirements
12          virtual ostream & Ostream (ostream & ) const;

13          // - local show/set functions
14          static void GlobalSetup (const int setNumOfChips,
15
16              const int setNumOfPipelinesPerChip,
17
18              const int setBlockSize,
19
20              Cube4 *setCube4);
21          virtual void LocalSetup (const int setChipIndex);
22          virtual void PerFrameSetup ();
23          // local computation functions
24          virtual void RunForOneClockCycle();

25          public:
26              GradZLinXPipeline *gradZLinXPipeline;
27              GradZLinXStageInputs inputs;
28              GradZLinXStageResults tempResults;
29              GradZLinXStageResults results;
30              FiFo<PerChipControlFlags> blockBeamControlFiFo;

31          protected:
32              static int    numOfChips, numOfPipelinesPerChip, blockSize;
33              static Cube4 *cube4;
34              int         chipIndex;

35              friend class GradZLinXPipeline;
36      };

37      #endif // _GradZLinXStage_h_
38      ::::::::::::::::::::
39      cube4/GradZLinYPipeline.C
40      ::::::::::::::::::::
41      // GradZLinYPipeline.C
42      // (c) Ingmar Bitter '97

43      // Copyright, Mitsubishi Electric Information Technology Center
44      // America, Inc., 1997, All rights reserved.

45      #include "GradZLinYPipeline.h"

void GradZLinYPipeline::Demo()
{
    GradZLinYPipeline gradZLinY;
    cout << endl << "Demo of class " << typeid(gradZLinY).name();
    cout << endl << "size : " << sizeof(GradZLinYPipeline) << " Bytes";
    cout << endl << "public member functions:";

55

```

```

197
    cout << endl << "GradZLinYPipeline gradZLinY; = " << gradZLinY;
    cout << endl << "End of demo of class " << typeid(gradZLinY).name() <<
5      endl;
} // Demo

///////////////////////////////
// constructors & destructors
10

// static first init
int GradZLinYPipeline::numOfChips           = 0;
int GradZLinYPipeline::numOfPipelinesPerChip = 0;
int GradZLinYPipeline::blockSize            = 0;
15 int GradZLinYPipeline::maxDataSetSizeX     = 256;
Cube4 *GradZLinYPipeline::cube4             = 0;

GradZLinYPipeline::GradZLinYPipeline()
{
20    // step delay for a y-step within a block
    int beamDelay = (maxDataSetSizeX
                      ) /
    (numOfChips*numOfPipelinesPerChip);

    // step delay for a y-step between blocks
    int blockBeamDelay = beamDelay * blockSize;

    beamGradientFiFo.SetSize(beamDelay);
    beamWeightsFiFo.SetSize(beamDelay);
    beamPerPipelineControlFlagsFiFo.SetSize(beamDelay);
    beamPerChipControlFlagsFiFo.SetSize(beamDelay);
30

    blockBeamGradientFiFo.SetSize(blockBeamDelay);
    blockBeamWeightsFiFo.SetSize(blockBeamDelay);
    blockBeamPerPipelineControlFlagsFiFo.SetSize(blockBeamDelay);
    blockBeamPerChipControlFlagsFiFo.SetSize(blockBeamDelay);
35 } // constructor

GradZLinYPipeline::~GradZLinYPipeline()
{
40 } // destructor

///////////////////////////////
// show/set data & data properties
45

ostream & GradZLinYPipeline::Ostream(ostream & os) const
{
    // append GradZLinYPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << " numOfChips        = " << numOfChips;
50    os << endl << " numOfPipelinesPerChip = " << numOfPipelinesPerChip;
}

```

```

198
    os << endl << " chipIndex      = " << chipIndex;

5   // return complete os
return os;

} // Ostream

10  /////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

15  void GradZLinYPipeline::GlobalSetup(const int setNumOfChips,
                                         const int setNumOfPipelinesPerChip,
                                         const int setBlockSize)
{
20    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize            = setBlockSize;
} // GlobalSetup

25  void GradZLinYPipeline::LocalSetup(const int setChipIndex,
                                         const int setPipelineIndex,
                                         GradZLinYStage & gradZLinYStage)
{
30    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.gz = &(gradZLinYStage.inputs.gz[pipelineIndex]);
    inputs.weightsXYZ = &(gradZLinYStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = gradZLinYStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(gradZLinYStage.inputs.perPipelineControlFlags[pipelineIndex]);

35    results.gz = &(gradZLinYStage.results.gz[pipelineIndex]);
    results.weightsXYZ = &(gradZLinYStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(gradZLinYStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(gradZLinYStage.results.perChipControlFlags);

40    cube4 = gradZLinYStage(cube4);
} // LocalSetup

45

50

```

```

199
void GradZLinYPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in GradZLinYStage
5
    // resize fifo's according to dataset size
    // step delay for a y-step within a block
    int beamDelay = (cube4->datasetSizeXYZ.X())
                                ) /
10   (numOfChips*numOfPipelinesPerChip);

    // step delay for a y-step between blocks
    int blockBeamDelay = beamDelay * blockSize;

    beamGradientFiFo.SetSize(beamDelay);
15   beamWeightsFiFo.SetSize(beamDelay);
    beamPerPipelineControlFlagsFiFo.SetSize(beamDelay);
    beamPerChipControlFlagsFiFo.SetSize(beamDelay);

    blockBeamGradientFiFo.SetSize(blockBeamDelay);
    blockBeamWeightsFiFo.SetSize(blockBeamDelay);
20   blockBeamPerPipelineControlFlagsFiFo.SetSize(blockBeamDelay);
    blockBeamPerChipControlFlagsFiFo.SetSize(blockBeamDelay);

    beamGradientFiFo.Preset(*(inputs.gz));
    beamWeightsFiFo.Preset(*(inputs.weightsXYZ));
25   beamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    blockBeamGradientFiFo.Preset(*(inputs.gz));
    blockBeamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    blockBeamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFla
30   gs));
    blockBeamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    readBigFiFoCounter = readSmallFiFoCounter =
        writeBigFiFoCounter = writeSmallFiFoCounter = -1;
} // PerFrameSetup
35

////////// local computation functions

void GradZLinYPipeline::RunForOneClockCycle()
40
{
    // reset counters
    if (inputs.perChipControlFlags->volumeStart ||
        ((readBigFiFoCounter == 0) &&
         (readSmallFiFoCounter == 0) &&
         (writeBigFiFoCounter == 0) &&
45       (writeSmallFiFoCounter == 0))) {
        readBigFiFoCounter = (cube4->datasetSizeXYZ.X())
}

```

50

55

200

```

) / (numOfChips*numOfPipelinesPerChip);
    readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;
5
    writeBigFiFoCounter = readBigFiFoCounter;
    writeSmallFiFoCounter = readSmallFiFoCounter;
}

10 /////////////////
// first read from FiFos into results register

// at start of block read from big FiFo
if (readBigFiFoCounter > 0) {
    blockBeamGradientFiFo.Read(outFiFo);
15    blockBeamWeightsFiFo.Read(*(results.weightsXYZ));
    blockBeamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlag
s));
}

20 blockBeamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
--readBigFiFoCounter;
}

// in middle and at end of block read from small FiFo
else if (readSmallFiFoCounter > 0) {
    beamGradientFiFo.Read(outFiFo);
25    beamWeightsFiFo.Read(*(results.weightsXYZ));
    beamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
    --readSmallFiFoCounter;
}

30 /////////////////
// now write to FiFos from inputs register

// at start and in middle of block write to small FiFo
35 if (writeSmallFiFoCounter > 0) {
    beamGradientFiFo.Write(*(inputs.gz));
    beamWeightsFiFo.Write(*(inputs.weightsXYZ));
    beamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Write(
40 *(inputs.perChipControlFlags));
    --writeSmallFiFoCounter;
}

// at end of block write to big FiFo of next chip
45 else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
    GradZLinYPipeline *next(&cube4->gradZLinY[c].gradZLinYPipeline[p]);
    next->blockBeamGradientFiFo.Write(*(inputs.gz));
}

```

50

55

```

201
    next-
>blockBeamWeightsFiFo.Write(*(inputs.weightsXYZ) );
    next-
5   >blockBeamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
    next-
>blockBeamPerChipControlFlagsFiFo.Write(*(inputs.perChipControlFlags));
    --writeBigFiFoCounter;
}

10

// Linear interpolation
// c = w(b-a)+a
//
// w=0 -> c=a
15 // w=1 -> c=b

if (cube4->cubeMode == Cube4Light) {
    // just pick current voxel (as if w=0)
    *(results.gz) = *(inputs.gz); // w=0;
20    // use weights later on in composing
}

else if (cube4->cubeMode == Cube4Classic) {
    // do real interpolation
    FixPointNumber a(*(inputs.gz)),
25        b(outFiFo);

    *(results.gz) = (ScalarGradient)((inputs.weightsXYZ->Y()) * (b - a)
+ a);
}
30 } // RunForOneClockCycle

///////////////////////////////
// internal utility functions

35 // end of GradZLinYPipeline.C
:::::::::::
cube4/GradZLinYPipeline.h
:::::::::::
// GradZLinYPipeline.h
40 // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45 #ifndef _GradZLinYPipeline_h_ // prevent multiple includes
#define _GradZLinYPipeline_h_

#include "Misc.h"
#include "Object.h"
#include "FiFo.h"

50

```

55

202

```

5      #include "Voxel.h"
#include "Coxel.h"
#include "Control.h"
#include "FixPointNumber.h"

10     class GradZLinYStage;
class Cube4;

15     typedef Vector3D<FixPointNumber> FixPointVector3D;

class GradZLinYPipelineInputs {
public: // pointers
    ScalarGradient *gz; // partially interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

20     class GradZLinYPipelineResults {
public: // pointers
    ScalarGradient *gz; // now also y-interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

25     class GradZLinYPipeline : virtual public Object {
public:

30         static void Demo ();

        // constructors & destructors
        GradZLinYPipeline ();
        ~GradZLinYPipeline ();

40         // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream & ) const;

        // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,
45            const int setNumOfPipelinesPerChip,
            const int setBlockSize);

        virtual void LocalSetup(const int setChipIndex,
50            const int setPipelineIndex,
            GradZLinYStage & gradZLinYStage);

```

55

```

203
    virtual void PerFrameSetup();
    // local computation functions
5     virtual void RunForOneClockCycle();

public:
    GradZLinYPipelineInputs inputs;
    GradZLinYPipelineResults results;

10    protected:
        Fifo<ScalarGradient> beamGradientFifo;
        Fifo<FixPointVector3D> beamWeightsFifo;
        Fifo<PerPipelineControlFlags> beamPerPipelineControlFlagsFifo;
        Fifo<PerChipControlFlags> beamPerChipControlFlagsFifo;

15
        Fifo<ScalarGradient> blockBeamGradientFifo;
        Fifo<FixPointVector3D> blockBeamWeightsFifo;
        Fifo<PerPipelineControlFlags> blockBeamPerPipelineControlFlagsFifo;
        Fifo<PerChipControlFlags> blockBeamPerChipControlFlagsFifo;

20    ScalarGradient outFifo;

    int readSmallFifoCounter, readBigFifoCounter;
    int writeSmallFifoCounter, writeBigFifoCounter;

25    static int numOFChips, numOFPipelinesPerChip, blockSize;
    static int maxDatasetSizeX;
    static Cube4 *cube4;
    int chipIndex, pipelineIndex, ,
30    };

    #include "GradZLinYStage.h"
    #include "Cube4.h"

35    #endif // _GradZLinYPipeline_h_
    ::::::::::::::
    cube4/GradZLinYStage.C
    ::::::::::::::
    // GradZLinYStage.C
    // (c) Ingmar Bitter '97

40    // Copyright, Mitsubishi Electric Information Technology Center
    // America, Inc., 1997, All rights reserved.

    #include "GradZLinYStage.h"

45    void GradZLinYStage::Demo()
    {
        GradZLinYStage gradZLinY;
        cout << endl << "Demo of class " << typeid(gradZLinY).name();
        cout << endl << "size : " << sizeof(GradZLinYStage) << " Bytes";
50        cout << endl << "public member functions:" ;
        cout << endl << "GradZLinYStage gradZLinY; = " << gradZLinY;
    }

```

```

      cout << endl << "End of demo of      class " << typeid(gradZLinY).name() <<
endl;
} // Demo
5

///////////////////////////////
// constructors & destructors

10   // static first init
int GradZLinYStage::numOfChips           = 0;
int GradZLinYStage::numOfPipelinesPerChip = 0;
Cube4 *GradZLinYStage::cube4 = 0;

GradZLinYStage::GradZLinYStage()
15 {
    gradZLinYPipeline = new GradZLinYPipeline [numOfPipelinesPerChip];
    results.gz = new ScalarGradient [numOfPipelinesPerChip+1]; // +1 for
communication
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
20 [numOfPipelinesPerChip];
} // defaultconstructor

GradZLinYStage::~GradZLinYStage()
25 {
    if (gradZLinYPipeline) { delete gradZLinYPipeline; gradZLinYPipeline=0;
}
    if (results.gz)        { delete results.gz;         results.gz=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
30    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor
35

///////////////////////////////
// show/set data & data properties

40   ostream & GradZLinYStage::Ostream(ostream & os) const
{
    // append GradZLinYStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;
45

    // return complete os
    return os;
}

50

```

55

```

205
} // Ostream

5
////////// show/set data & data properties
//
// - local show/set functions

10 void GradZLinYStage::GlobalSetup(const int setNumOfChips,
                                    const int setNumOfPipelinesPerChip,
                                    const int setBlockSize,
15
                                    Cube4 *setCube4)
{
    numOfChips      = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    GradZLinYPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
20
                                    setBlockSize);
    cube4 = setCube4;
} // GlobalSetup

25 void GradZLinYStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinYPipeline[p].LocalSetup(chipIndex,p,*this);
30
    }
} // LocalSetup

35 void GradZLinYStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.gz[p] = 0;
        results.weightsXYZ[p](0,0,0);
40
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

    for (p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinYPipeline[p].PerFrameSetup();
45
    }

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup
50

```

```

5   //////////////////////////////////////////////////////////////////
// local computation functions

void GradZLinYStage::RunForOneClockCycle()
{
    // communication

    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinYPipeline[p].RunForOneClockCycle();
    }
15 } // RunForOneClockCycle

////////////////////////////////////////////////////////////////
// internal utility functions

20

// end of GradZLinYStage.C
:::::::::::
cube4/GradZLinYStage.h
:::::::::::
25 // GradZLinYStage.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30 #ifndef _GradZLinYStage_h_      // prevent multiple includes
#define _GradZLinYStage_h_

35 #include "Misc.h"
#include "Object.h"
#include "Vector3D.h"
#include "Voxel.h"
#include "Control.h"
#include "GradZLinYPipeline.h"
#include "FixPointNumber.h"
40 #include "Cube4.h"

class Cube4;

class GradZLinYStageInputs {
45 public: // pointers
    ScalarGradient *gz; // partially interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags     *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

50

```

207

```

5      class GradZLinYStageResults {
public: // arrays
    ScalarGradient *gz; // now also y-interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

10
15     class GradZLinYStage : virtual public Object {
public:

    static void Demo ();

// constructors & destructors
GradZLinYStage ();
~GradZLinYStage ();

20
25     // show/set data & data properties
// - class Object requirements
virtual ostream & Ostream (ostream & ) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
                        const int setNumOfPipelinesPerChip,
                        const int setBlockSize,
                        Cube4 *setCube4);
virtual void LocalSetup (const int setChipIndex);
virtual void PerFrameSetup ();
// local computation functions
virtual void RunForOneClockCycle();

30
35     public:
        GradZLinYPipeline *gradZLinYPipeline;
        GradZLinYStageInputs inputs;
        GradZLinYStageResults results;

40     protected:
        static int numOfChips, numOfPipelinesPerChip;
        static Cube4 *cube4;
        int chipIndex;

45     friend class GradZLinYPipeline;
};

50
55 #endif // _GradZLinYStage_h_
:::::::::::
cube4/GradZLinZPipeline.C
:::::::::::
// GradZLinZPipeline.C

```

55

208

```

// (c) Ingmar Bitter '97

5 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "GradZLinZPipeline.h"

void GradZLinZPipeline::Demo()
10 {
    GradZLinZPipeline gradZLinZ;
    cout << endl << "Demo of class " << typeid(gradZLinZ).name();
    cout << endl << "size : " << sizeof(GradZLinZPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "GradZLinZPipeline gradZLinZ; = " << gradZLinZ;
15    cout << endl << "End of demo of class " << typeid(gradZLinZ).name() <<
        endl;
} // Demo

20 /////////////////////////////////
// constructors & destructors

// static first init
int GradZLinZPipeline::numOfChips          = 0;
int GradZLinZPipeline::numOfPipelinesPerChip = 0;
25 int GradZLinZPipeline::blockSize          = 0;
int GradZLinZPipeline::maxDatasetSizeX      = 256;
int GradZLinZPipeline::maxDatasetSizeY      = 256;
Cube4 *GradZLinZPipeline::cube4             = 0;

GradZLinZPipeline::GradZLinZPipeline()
30 {
    // step delay for a z-step within a block
    int blockSliceDelay = (maxDatasetSizeX*blockSize
                           ) /
                           (numOfChips*numOfPipelinesPerChip);

    // step delay for a z-step between blocks
    int volumeSliceDelay = (maxDatasetSizeX*maxDatasetSizeY
                           )
                           / (numOfChips*numOfPipelinesPerChip);

40    blockSliceGradientFiFo.SetSize(blockSliceDelay);
    blockSliceWeightsFiFo.SetSize(blockSliceDelay);
    blockSlicePerPipelineControlFlagsFiFo.SetSize(blockSliceDelay);
    blockSlicePerChipControlFlagsFiFo.SetSize(blockSliceDelay);

    volumeSliceGradientFiFo.SetSize(volumeSliceDelay);
    volumeSliceWeightsFiFo.SetSize(volumeSliceDelay);
45    volumeSlicePerPipelineControlFlagsFiFo.SetSize(volumeSliceDelay);
    volumeSlicePerChipControlFlagsFiFo.SetSize(volumeSliceDelay);
} // constructor

50

```

55

```

GradZLinZPipeline::~GradZLinZPipeline()
{
} // destructor

///////////////////////////////
// show/set data & data properties
10

ostream & GradZLinZPipeline::Ostream(ostream & os) const
{
    // append GradZLinZPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
15    os << endl << "numOfChips" = " " << numOfChips;
    os << endl << "numOfPipelinesPerChip" = " " << numOfPipelinesPerChip;
    os << endl << "chipIndex" = " " << chipIndex;

    // return complete os
20    return os;
}

} // Ostream

///////////////////////////////
25 // show/set data & data properties
//
// - local show/set functions

30 void GradZLinZPipeline::GlobalSetup(const int setNumOfChips,
                                       const int setNumOfPipelinesPerChip,
                                       const int setBlockSize)
{
35    numOfChips = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize = setBlockSize;
} // GlobalSetup

40 void GradZLinZPipeline::LocalSetup(const int setChipIndex,
                                       const int setPipelineIndex,
                                       GradZLinZStage & gradZLinZStage)
45 {
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.gz = &(gradZLinZStage.inputs.gz[pipelineIndex]);
    inputs.weightsXYZ = &(gradZLinZStage.inputs.weightsXYZ[pipelineIndex]);
50
}

```

```

210
inputs.perChipControlFlags
    = gradZLinZStage.inputs.perChipControlFlags;
inputs.perPipelineControlFlags
    = &(gradZLinZStage.inputs.perPipelineControlFlags[pipelineIndex]);


results.gz = &(gradZLinZStage.results.gz[pipelineIndex]);
results.weightsXYZ = &(gradZLinZStage.results.weightsXYZ[pipelineIndex]);
results.perPipelineControlFlags
    = &(gradZLinZStage.results.perPipelineControlFlags[pipelineIndex]);
results.perChipControlFlags
    = &(gradZLinZStage.results.perChipControlFlags);

cube4 = gradZLinZStage(cube4;
} // LocalSetup
15

void GradZLinZPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in GradZLinZStage

20    // resize fifo's according to dataset size
    // step delay for a z-step within a block
    int blockSliceDelay = (cube4->datasetSizeXYZ.X()*blockSize
                           / (numOfChips*numOfPipelinesPerChip));
                           ) / (numOfChips*numOfPipelinesPerChip);

    // step delay for a z-step between blocks
    int volumeSliceDelay = (cube4->datasetSizeXYZ.X()*cube4-
                           >datasetSizeXYZ.Y())
                           / (numOfChips*numOfPipelinesPerChip);
                           )

30    blockSliceGradientFiFo.SetSize(blockSliceDelay);
    blockSliceWeightsFiFo.SetSize(blockSliceDelay);
    blockSlicePerPipelineControlFlagsFiFo.SetSize(blockSliceDelay);
    blockSlicePerChipControlFlagsFiFo.SetSize(blockSliceDelay);

35    volumeSliceGradientFiFo.SetSize(volumeSliceDelay);
    volumeSliceWeightsFiFo.SetSize(volumeSliceDelay);
    volumeSlicePerPipelineControlFlagsFiFo.SetSize(volumeSliceDelay);
    volumeSlicePerChipControlFlagsFiFo.SetSize(volumeSliceDelay);

40    blockSliceGradientFiFo.Preset(*(inputs.gz));
    blockSliceWeightsFiFo.Preset(*(inputs.weightsXYZ));
    blockSlicePerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlF1
ags));
    blockSlicePerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

45    volumeSliceGradientFiFo.Preset(*(inputs.gz));
    volumeSliceWeightsFiFo.Preset(*(inputs.weightsXYZ));
    volumeSlicePerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlF
lags));
    volumeSlicePerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));
}
50

```

55

211

```

      readBigFiFoCounter = readSmallFiFoCounter =
      writeBigFiFoCounter = writeSmallFiFoCounter = -1;
5   } // PerFrameSetup

      /////////////////
      // local computation functions
10
void GradZLinZPipeline::RunForOneClockCycle()
{
    // reset counters
    if (inputs.perChipControlFlags->volumeStart ||
        (readBigFiFoCounter == 0) &&
15      (readSmallFiFoCounter == 0) &&
        (writeBigFiFoCounter == 0) &&
        (writeSmallFiFoCounter == 0))) {

        readBigFiFoCounter = (cube4->datasetSizeXYZ.X()*blockSize
20
        / (numOfChips*numOfPipelinesPerChip);
        readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;

        writeBigFiFoCounter = readBigFiFoCounter;
        writeSmallFiFoCounter = readSmallFiFoCounter;
25
    }

    ///////////////
    // first read from Fifos into results register

    // at start of block read from big Fifo
30
    if (readBigFiFoCounter > 0) {
        volumeSliceGradientFiFo.Read(outFiFo);
        volumeSliceWeightsFiFo.Read(*(results.weightsXYZ));

        volumeSlicePerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFl
ags));
35
        volumeSlicePerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
        --readBigFiFoCounter;
        // if (chipIndex == 0 && pipelineIndex == 0)
        cout<<"R"<<*(results.voxel)<<volumeSliceGradientFiFo<<endl;
    }
40
    // in middle and at end of block read from small Fifo
    else if (readSmallFiFoCounter > 0) {
        blockSliceGradientFiFo.Read(outFiFo);
        blockSliceWeightsFiFo.Read(*(results.weightsXYZ));

        blockSlicePerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFla
gs));
45
        blockSlicePerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
    }
50
}

```

55

```

212
--readSmallFiFoCounter;
// if (chipIndex == 0 && pipelineIndex == 0) cout<<"r";
}

///////////////////////////////
// now write to FiFos from inputs register

10 // at start and in middle of block write to small FiFo
if (writeSmallFiFoCounter > 0) {
    blockSliceGradientFiFo.Write(*(inputs.gz));
    blockSliceWeightsFiFo.Write(*(inputs.weightsXYZ) );
}

15 blockSlicePerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFla
gs));
    blockSlicePerChipControlFlagsFiFo.Write(
*(inputs.perChipControlFlags));
    --writeSmallFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0) cout<<"w";
}

20 // at end of block write to big FiFo of next chip
else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
25    GradZLinZPipeline *next(&cube4->gradZLinZ[c].gradZLinZPipeline[p]);
    next->volumeSliceGradientFiFo.Write(*(inputs.gz));
    next->volumeSliceWeightsFiFo.Write(*(inputs.weightsXYZ) );
    next-
>volumeSlicePerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags))
;

30    next-
>volumeSlicePerChipControlFlagsFiFo.Write(*(inputs.perChipControlFlags));
    --writeBigFiFoCounter;
    //if (chipIndex == 0 && pipelineIndex == 0) cout<<"W";
}

35 // Linear interpolation
// c = w(b-a)+a
//
// w=0 -> c=a
// w=1 -> c=b

40 if (cube4->cubeMode == Cube4Light) {
    // just pick current voxel (as if w=0)
    *(results.gz) = *(inputs.gz); // w=0;
    // use weights later on in composing
}

45 else if (cube4->cubeMode == Cube4Classic) {
    // do real interpolation
    a(*(inputs.gz));
    b(outFiFo);
}

```

50

55

213

```

5      *(results.gz) = (ScalarGradient) ((inputs.weightsXYZ->Z()) * (b - a)
+ a);
    }
} // RunForOneClockCycle

10
// internal utility functions

// end of GradZLinZPipeline.C
:::::::::::
cube4/GradZLinZPipeline.h
:::::::::::
15
// GradZLinZPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

20
#ifndef _GradZLinZPipeline_h_ // prevent multiple includes
#define _GradZLinZPipeline_h_

#include "Misc.h"
#include "Object.h"
25
#include "FiFo.h"
#include "Voxel.h"
#include "Coxel.h"
#include "Control.h"
#include "FixPointNumber.h"

30
typedef Vector3D<FixPointNumber> FixPointVector3D;

class GradZLinZStage;
class Cube4;

35
class GradZLinZPipelineInputs {
public: // pointers
    ScalarGradient *gz; // on voxel grid
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags    *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

40

class GradZLinZPipelineResults {
public: // pointers
    ScalarGradient *gz; // now z-interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
45
    PerChipControlFlags    *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

```

50

55

```

5   class GradZLinZPipeline : virtual public Object {
public:

    static void      Demo ();

    // constructors & destructors
10  GradZLinZPipeline ();
    ~GradZLinZPipeline ();

    // show/set data & data properties
    // - class Object requirements
15  virtual ostream & Ostream (ostream & ) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
20
        const int setNumOfPipelinesPerChip,
        const int setBlockSize);

    virtual void LocalSetup(const int setChipIndex,
25
        const int setPipelineIndex,
        GradZLinZStage & gradZLinZStage);
    virtual void PerFrameSetup();
    // local computation functions
30  virtual void RunForOneClockCycle();

public:
    GradZLinZPipelineInputs inputs;
    GradZLinZPipelineResults results;

35  protected:
    Fifo<ScalarGradient>          blockSliceGradientFifo;
    Fifo<FixPointVector3D>         blockSliceWeightsFifo;
    Fifo<PerPipelineControlFlags>  blockSlicePerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags>      blockSlicePerChipControlFlagsFifo;

40  Fifo<ScalarGradient>          volumeSliceGradientFifo;
    Fifo<FixPointVector3D>         volumeSliceWeightsFifo;
    Fifo<PerPipelineControlFlags>  volumeSlicePerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags>      volumeSlicePerChipControlFlagsFifo;

45  int   readSmallFifoCounter,  readBigFifoCounter;
    int   writeSmallFifoCounter,  writeBigFifoCounter;

    ScalarGradient outFifo;
    FixPointNumber a,b;

50  static int   numOfChips,  numOfPipelinesPerChip, blockSize;

```

```

215
static int           maxDatasetSizeX,maxDatasetSizeY;
static Cube4 *cube4;
int      chipIndex, pipelineIndex;

5   friend class Cube4;

friend class Cube4;

}; // class GradZLinZPipeline

10 #include "GradZLinZStage.h"
#include "Cube4.h"

#ifndef      _GradZLinZPipeline_h_
::::::::::::::::::::
cube4/GradZLinZStage.C
15 ::::::::::::::::::::
// GradZLinZStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.
20

#include "GradZLinZStage.h"

void GradZLinZStage::Demo()
{
25   GradZLinZStage gradZLinZ;
   cout << endl << "Demo of class " << typeid(gradZLinZ).name();
   cout << endl << "size : " << sizeof(GradZLinZStage) << " Bytes";
   cout << endl << "public member functions:";
   cout << endl << "GradZLinZStage gradZLinZ; = " << gradZLinZ;
   cout << endl << "End of demo of class "<< typeid(gradZLinZ).name() <<
30   endl;
} // Demo

::::::::::::::::::
// constructors & destructors
35

// static first init
int GradZLinZStage::numOfChips          = 0;
int GradZLinZStage::numOfPipelinesPerChip = 0;
Cube4 *GradZLinZStage::cube4 = 0;

40 GradZLinZStage::GradZLinZStage()
{
   gradZLinZPipeline = new GradZLinZPipeline [numOfPipelinesPerChip];
   results.gz = new ScalarGradient [numOfPipelinesPerChip];
   results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
   results.perPipelineControlFlags = new PerPipelineControlFlags
45 [numOfPipelinesPerChip];
} // defaultconstructor

```

50

55

```

216
GradZLinZStage::~GradZLinZStage()
{
    if (gradZLinZPipeline) { delete gradZLinZPipeline; gradZLinZPipeline=0;
5     }
    if (results.gz)         { delete results.gz;           results.gz=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

15 //////////////////////////////////////////////////////////////////
// show/set data & data properties

ostream & GradZLinZStage::Ostream(ostream & os) const
{
20   // append GradZLinZStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips      = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex       = " << chipIndex;

25   // return complete os
    return os;
}

} // Ostream

30 //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

35 void GradZLinZStage::GlobalSetup(const int setNumOfChips,
                                   
                                    const int setNumOfPipelinesPerChip,
                                   
                                    const int setBlockSize,
40
                                    Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    GradZLinZPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
45
                                    setBlockSize);
    cube4 = setCube4;
} // GlobalSetup

50

```

55

```

5   void GradZLinZStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        gradZLinZPipeline[p].LocalSetup(chipIndex,p,*this);
    }
10 } // LocalSetup

void GradZLinZStage::PerFrameSetup()
{
15     int p;
     // reset pipeline registers
     for (p=0; p<numOfPipelinesPerChip; ++p) {
         results.gz[p] = 0;
         results.weightsXYZ[p](0,0,0);
         results.perPipelineControlFlags[p].Reset();
     }
20     results.perChipControlFlags.Reset();

     for (p=0; p<numOfPipelinesPerChip; ++p) {
         gradZLinZPipeline[p].PerFrameSetup();
     }
25     // print debug info
     //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

30 //////////////////////////////////////////////////////////////////
// local computation functions

void GradZLinZStage::RunForOneClockCycle()
{
35     // communication

     // computation
     for (int p=0; p<numOfPipelinesPerChip; ++p) {
         gradZLinZPipeline[p].RunForOneClockCycle();
     }
40 } // RunForOneClockCycle

45 //////////////////////////////////////////////////////////////////
// internal utility functions

// end of GradZLinZStage.C
:::::::::::
cube4/GradZLinZStage.h
:::::::::::
50

```

```

// GradZLinZStage.h
// (c) Ingmar Bitter '97
5
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _GradZLinZStage_h_      // prevent multiple includes
10 #define _GradZLinZStage_h_

#include "Misc.h"
#include "Object.h"
#include "Vector3D.h"
#include "Voxel.h"
15 #include "Control.h"
#include "GradZLinZPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

20 class Cube4;

class GradZLinZStageInputs {
public: // pointers
    ScalarGradient *gz; // on voxel grid
    Vector3D<FixPointNumber> *weightsXYZ;
25    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

30 class GradZLinZStageResults {
public: // arrays
    ScalarGradient *gz; // now z-interpolated
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
35    PerPipelineControlFlags *perPipelineControlFlags;
};

class GradZLinZStage : virtual public Object {
public:
40    static void Demo ();

        // constructors & destructors
        GradZLinZStage ();
        ~GradZLinZStage ();

45    // show/set data & data properties
        // - class Object requirements
        virtual ostream & ostream (ostream &) const;

50    // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,

```

```

    const int setNumOfPipelinesPerChip,
5      const int setBlockSize,
        Cube4 *setCube4);
        virtual void LocalSetup (const int setChipIndex);
        virtual void PerFrameSetup ();
10     // local computation functions
        virtual void RunForOneClockCycle();

public:
    GradZLinZPipeline *gradZLinZPipeline;
    GradZLinZStageInputs inputs;
15    GradZLinZStageResults results;

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
20    int          chipIndex;

    friend class GradZLinZPipeline;
};

25 #endif // _GradZLinZStage_h_
:::::::cube4/GradZPipeline.C::::::::::
// GradZPipeline.C
// (c) Ingmar Bitter '97
30 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "GradZPipeline.h"

35 void GradZPipeline::Demo()
{
    GradZPipeline gradZ;
    cout << endl << "Demo of class " << typeid(gradZ).name();
    cout << endl << "size : " << sizeof(GradZPipeline) << " Bytes";
    cout << endl << "public member functions:";
40    cout << endl << "GradZPipeline gradZ; = " << gradZ;
    cout << endl << "End of demo of class " << typeid(gradZ).name() << endl;
} // Demo

45 /////////////////////////////////
// constructors & destructors

// static first init
int GradZPipeline::numOfChips           = 0;
int GradZPipeline::numOfPipelinesPerChip = 0;
50

```

```

5     GradZPipeline::GradZPipeline()
6     {
7     } // constructor

10    GradZPipeline::~GradZPipeline()
11    {
12    } // destructor

15    //////////////////////////////////////////////////////////////////
16    // show/set data & data properties

17    ostream & GradZPipeline::Ostream(ostream & os) const
18    {
19        // append GradZPipeline info to os
20        os << typeid(*this).name() << "@" << (void *) this;
21        os << endl << "numOfChips" = " " << numOfChips;
22        os << endl << "numOfPipelinesPerChip" = " " << numOfPipelinesPerChip;
23        os << endl << "chipIndex" = " " << chipIndex;

24        // return complete os
25        return os;
26    } // Ostream

30    //////////////////////////////////////////////////////////////////
31    // show/set data & data properties
32    //
33    // - local show/set functions

35    void GradZPipeline::GlobalSetup(const int setNumOfChips,
36
37        const int setNumOfPipelinesPerChip)
38    {
39        numOfChips = setNumOfChips;
40        numOfPipelinesPerChip = setNumOfPipelinesPerChip;
41    } // GlobalSetup

45    void GradZPipeline::LocalSetup(const int setChipIndex,
46
47        const int setPipelineIndex,
48
49        GradZStage & gradZStage)
50    {
51        chipIndex = setChipIndex;
52        pipelineIndex = setPipelineIndex;

```

```

221
    inputs.voxel0 =
5     &(gradZStage.inputs.voxel0[pipelineIndex]);
    inputs.voxel1 = &(gradZStage.inputs.voxel1[pipelineIndex]);
    inputs.weightsXYZ = &(gradZStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = gradZStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(gradZStage.inputs.perPipelineControlFlags[pipelineIndex]);
10
    results.weightsXYZ = &(gradZStage.results.weightsXYZ[pipelineIndex]);
    results.gz = &(gradZStage.results.gz[pipelineIndex]);
    results.perPipelineControlFlags
        = &(gradZStage.results.perPipelineControlFlags[pipelineIndex]);
15    results.perChipControlFlags
        = &(gradZStage.results.perChipControlFlags);
} // LocalSetup

20 void GradZPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in GradZStage
    delayVoxel0 = delayVoxel0again = 0;
    delayWeightsXYZ(0,0,0);
    delayPerPipelineControlFlags.Reset();
25    delayPerChipControlFlags.Reset();
} // PerFrameSetup

30
// local computation functions
void GradZPipeline::RunForOneClockCycle()
{
    /*
35     / (z)
     /voxB (seen late (recently) during processing => take from memCtrl )
     /
     /voxA (seen early (long ago) during processing => take from FiFo1 )
     /
     +-----> (x)
     |
     |
     v(y)
40
     */
     ScalarGradient voxB = delayVoxel0again.raw16bit;
     ScalarGradient voxA = inputs.voxel1->raw16bit;

     *(results.gz) = voxB - voxA;
50

```

```

222
// complete the time-lineup

5      *(results.weightsXYZ) = delayWeightsXYZ;
      *(results.perPipelineControlFlags) = delayPerPipelineControlFlags;
      if (pipelineIndex == 0)
          *(results.perChipControlFlags) = delayPerChipControlFlags;

10     delayVoxel0again = delayVoxel0;
      delayVoxel0 = *(inputs.voxel0);

      delayWeightsXYZ = *(inputs.weightsXYZ);
      delayPerPipelineControlFlags = *(inputs.perPipelineControlFlags);
      delayPerChipControlFlags = *(inputs.perChipControlFlags);

15 } // RunForOneClockCycle

///////////////////////////////
20 // internal utility functions

// end of GradZPipeline.C
:::::::::::
cube4/GradZPipeline.h
:::::::::::
// GradZPipeline.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
30 // America, Inc., 1997, All rights reserved.

#ifndef _GradZPipeline_h_      // prevent multiple includes
#define _GradZPipeline_h_

35 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Coxel.h"
#include "Control.h"
#include "FixPointNumber.h"

40 class GradZStage;
class Cube4;

class GradZPipelineInputs {
45 public: // pointers
    Voxel *voxel0;
    Voxel *voxel1;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
50 };

```

```

5      class GradZPipelineResults {
public: // pointers
       ScalarGradient *gz; // on voxel grid
       Vector3D<FixPointNumber> *weightsXYZ;
       PerChipControlFlags    *perChipControlFlags;
       PerPipelineControlFlags *perPipelineControlFlags;
} ;

10

15     class GradZPipeline : virtual public Object {
public:

20         static void      Demo ();
               // constructors & destructors
               GradZPipeline ();
               ~GradZPipeline ();

25         // show/set data & data properties
         // - class Object requirements
         virtual ostream & Ostream (ostream & ) const;

         // - local show/set functions
         static void GlobalSetup (const int setNumOfChips,
                                 const int setNumOfPipelinesPerChip);

30         virtual void LocalSetup(const int setChipIndex,
                                 const int setPipelineIndex,
                                 GradZStage & gradZStage);
         virtual void PerFrameSetup();
         // local computation functions
         virtual void RunForOneClockCycle();

35         public:
               GradZPipelineInputs inputs;
               GradZPipelineResults results;

40         protected:
               Voxel delayVoxel0;
               Voxel delayVoxel0again;
               Vector3D<FixPointNumber> delayWeightsXYZ;
               PerChipControlFlags    delayPerChipControlFlags;
               PerPipelineControlFlags delayPerPipelineControlFlags;
               static int   numOfChips, numOfPipelinesPerChip;
               static Cube4 *cube4;
               int        chipIndex, pipelineIndex;

45         friend class Cube4;
};

50

```

```

5      #include "GradZStage.h"
      #include "Cube4.h"

10     #endif      // _GradZPipeline_h_
      ::::::::::::::
      cube4/GradZStage.C
      ::::::::::::::
10    // GradZStage.C
10    // (c) Ingmar Bitter '97

15    // Copyright, Mitsubishi Electric Information Technology Center
15    // America, Inc., 1997, All rights reserved.

15    #include "GradZStage.h"

15    void GradZStage::Demo()
15    {
15        GradZStage gradZ;
15        cout << endl << "Demo of class " << typeid(gradZ).name();
20        cout << endl << "size : " << sizeof(GradZStage) << " Bytes";
20        cout << endl << "public member functions:";
20        cout << endl << "GradZStage gradZ; = " << gradZ;
20        cout << endl << "End of demo of class " << typeid(gradZ).name() << endl;
20    } // Demo

25
25    /////////////////////////////////
25    // constructors & destructors

30    // static first init
30    int GradZStage::numOfChips          = 0;
30    int GradZStage::numOfPipelinesPerChip = 0;
30    Cube4 *GradZStage::cube4 = 0;

35    GradZStage::GradZStage()
35    {
35        gradZPipeline = new GradZPipeline [numOfPipelinesPerChip];
35        results.gz = new ScalarGradient [numOfPipelinesPerChip];
35        results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
35        results.perPipelineControlFlags = new PerPipelineControlFlags
35        [numOfPipelinesPerChip];
35    } // defaultconstructor

40

40    GradZStage::~GradZStage()
40    {
40        if (gradZPipeline) { delete gradZPipeline; gradZPipeline=0; }
40        if (results.gz) { delete results.gz; results.gz=0; }
45        if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
45        }
45        if (results.perPipelineControlFlags) {
45            delete results.perPipelineControlFlags;

```

50

55

```

      results.perPipelineControlFlags=0;
5    }
} // destructor

10
// show/set data & data properties

ostream & GradZStage::Ostream(ostream & os) const
{
15  // append GradZStage info to os
  os << typeid(*this).name() << "@" << (void *) this;
  os << endl << "  numOfChips          = " << numOfChips;
  os << endl << "  numOfPipelinesPerChip = " << numofPipelinesPerChip;
  os << endl << "  chipIndex           = " << chipIndex;

20  // return complete os
  return os;
}

} // Ostream

25
// show/set data & data properties
//
// - local show/set functions

30 void GradZStage::GlobalSetup(const int setNumOfChips,
                               const int setNumOfPipelinesPerChip,
                               Cube4 *setCube4)
35 {
  numOfChips          = setNumOfChips;
  numofPipelinesPerChip = setNumOfPipelinesPerChip;
  GradZPipeline::GlobalSetup(numOfChips, numofPipelinesPerChip);
  cube4 = setCube4;
40 }

} // GlobalSetup

void GradZStage::LocalSetup(const int setChipIndex)
{
45  chipIndex = setChipIndex;
  for (int p=0; p<numOfPipelinesPerChip; ++p) {
    gradZPipeline[p].LocalSetup(chipIndex,p,*this);
  }
} // LocalSetup

50 void GradZStage::PerFrameSetup()

```

226

```

1      int p;
2      // reset pipeline registers
3      for (p=0; p<numOfPipelinesPerChip; ++p) {
4          results.gz[p] = 0;
5          results.weightsXYZ[p](0,0,0);
6          results.perPipelineControlFlags[p].Reset();
7      }
8      results.perChipControlFlags.Reset();
9
10     for (p=0; p<numOfPipelinesPerChip; ++p) {
11         gradZPipeline[p].PerFrameSetup();
12     }
13
14     // print debug info
15     //static bool first(true); if (first) { cout<<this<<endl; first=false; }
16 } // PerFrameSetup
17
18 ///////////////////////////////////////////////////////////////////
19 // local computation functions
20
21 void GradZStage::RunForOneClockCycle()
22 {
23     // computation
24     for (int p=0; p<numOfPipelinesPerChip; ++p) {
25         gradZPipeline[p].RunForOneClockCycle();
26     }
27 } // RunForOneClockCycle
28
29 ///////////////////////////////////////////////////////////////////
30 // internal utility functions
31
32 // end of GradZStage.C
33 ::::::::::::::::::::
34 cube4/GradZStage.h
35 ::::::::::::::::::::
36 // GradZStage.h
37 // (c) Ingmar Bitter '97
38
39 // Copyright, Mitsubishi Electric Information Technology Center
40 // America, Inc., 1997, All rights reserved.
41
42 #ifndef _GradZStage_h_ // prevent multiple includes
43 #define _GradZStage_h_
44
45 #include "Misc.h"
46 #include "Object.h"
47 #include "Voxel.h"
48 #include "Control.h"
49 #include "GradZPipeline.h"
50

```

55

```

227
5      #include "FixPointNumber.h"
6      #include "Cube4.h"
7
8      class Cube4;
9
10     class GradZStageInputs {
11     public: // pointers
12         Voxel *voxel0; // from memCtrl
13         Voxel *voxel1; // from SliceVoxelFiFo1
14         Vector3D<FixPointNumber> *weightsXYZ; // from SliceVoxelFiFo0
15         PerChipControlFlags      *perChipControlFlags;
16         PerPipelineControlFlags *perPipelineControlFlags;
17     };
18
19
20     class GradZStageResults {
21     public: // arrays
22         ScalarGradient *gz; // on voxel grid
23         Vector3D<FixPointNumber> *weightsXYZ;
24         PerChipControlFlags      perChipControlFlags;
25         PerPipelineControlFlags *perPipelineControlFlags;
26     };
27
28     class GradZStage : virtual public Object {
29     public:
30         static void      Demo ();
31
32         // constructors & destructors
33         GradZStage ();
34         ~GradZStage ();
35
36         // show/set data & data properties
37         // - class Object requirements
38         virtual ostream & Ostream (ostream & ) const;
39
40         // - local show/set functions
41         static void GlobalSetup (const int setNumOfChips,
42
43             const int setNumOfPipelinesPerChip,
44
45             Cube4 *setCube4);
46         virtual void LocalSetup (const int setChipIndex);
47         virtual void PerFrameSetup ();
48         // local computation functions
49         virtual void RunForOneClockCycle();
50
51     public:
52         GradZPipeline *gradZPipeline;
53         GradZStageInputs inputs;
54         GradZStageResults results;
55

```

228

```

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    int          chipIndex; // only for debugging purpose

    friend class Cube4;
};

10 #endif // _GradZStage_h_
::::::
cube4/Light.C
::::::
// Light.C
15 // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

20 #include "Light.h"
#include <strstream.h> // strstream

void Light::Demo()
{
    Light light;
25   cout << endl << "Demo of class " << typeid(light).name();
    cout << endl << "size : " << sizeof(Light) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "Light light;" << light;
    // ...
30   cout << endl << "End of demo of class " << typeid(light).name() << endl;
} // Demo

///////////////////////////////
35 // constructors & destructors
Light::Light()
    : direction(0,0,0), intensity(0,0,0), sharpness(1)
{
} // constructor

40 Light::Light(double Px, double Py, double Pz,
               double Ir, double Ig, double Ib, int
setSharpness)
    : direction(Px,Py,Pz), intensity(Ir,Ig,Ib), sharpness(setSharpness)
{
} // constructor

45 Light::Light(const Vector3D<double> & setDirection,
               const
Vector3D<double> & setIntensity,
               const int
setSharpness)

```

55

```

229
    : direction(setDirection),           intensity(setIntensity),
sharpness(setSharpness)
5
} // constructor

Light::Light(const Light & src)
    : direction(src.direction), intensity(src.intensity), sharpness(src.sharpness)
10
} // constructor

15
// show/set data & data properties
// 
ostream & Light::Ostream(ostream & os) const
{
    // append Light info to os
    os << "dir" << direction << " Irgb" << intensity << " s: " << sharpness;
20
    // return complete os
    return os;

} // Ostream
25

Light & Light::operator () (const double Px, const double Py, const double Pz)
{
    direction(Px,Py,Pz);
    intensity(1,1,1);
30
    sharpness = 1;
    return *this;
} // operator ()

35
Light & Light::operator () (const double Px, const double Py, const double Pz,
                           const double Ir, const double Ig, const double Ib,
                           const int setSharpness)
{
    direction(Px,Py,Pz);
    intensity(Ir,Ig,Ib);
    sharpness = setSharpness;
40
    return *this;
} // operator ()

45
Light & Light::operator = (const int k)
{
    direction(k,k,k);
    intensity(0,0,0);
50

```

230

```

5           sharpness = 1;
           return *this;
5 // operator ()

Light & Light::operator () (const char * str)
{
    strstream lightVectorData;
    char buff[200];

    if (str[0] == '\"') {
        // lightVector in double quotes
        // copy only part between double quotes
        strcpy(buff, &str[1]);
        buff[strcspn(buff, '\"')] = 0;
        lightVectorData << buff;
    }
    else {
        // plain lightVector
        // copy completely
        lightVectorData << str;
    }
    lightVectorData >> (*this);
    return *this;
26 } // operator ()

istream & operator >> (istream & is, Light & light)
{
30     double Px; double Py; double Pz;
     double Ir=1; double Ig=1; double Ib=1;
     int setSharpness=1;

     is >> Px >> Py >> Pz;
     if (is) is >> Ir >> Ig >> Ib;
     if (is) is >> setSharpness;

     light(Px,Py,Pz, Ir,Ig,Ib, setSharpness);

40     return is;
} // operator >>

45 Vector3D<double> Light::Direction()
{
    return direction;
} // Direction

50 Vector3D<double> Light::Intensity()
{
    return intensity;
} // Intensity

```

55

```

int Light::Sharpness()
{
    return sharpness;
} // Sharpness

Light & Light::TransformDirection(const Matrix4x4<int> & M)
{
    direction = M * direction;
    return *this;
} // TransformDirection

// end of Light.C
:::::::::::
cube4/Light.h
:::::::::::
// Light.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

#ifndef _Light_h_ // prevent multiple includes
#define _Light_h_
#include "Object.h"
#include "Vector3D.h"
#include "Matrix4x4.h"

class Light : virtual public Object {
public:
    static void Demo ();

    // constructors & destructors

    /*inline*/ Light();
    /*inline*/ Light(double Px, double Py, double Pz,
Ig=1, double Ib=1,                               double Ir=1, double
        /*inline*/     Light(const Vector3D<double> & setPos,           int setSharpness=1);
        Vector3D<double> & setIntensity,                                     const
        setSharpness);                                                 const int
    /*inline*/     Light(const Light & src);
    // show/set data & data properties

    virtual /*inline*/ Light & operator () (const double Px,

```

50

55

```

      const double Px,
5
      const double Pz,
      const double Ir,
      const double Ig,
      const double Ib,
      const int setSharpness); // set
function
    virtual /*inline*/ Light & operator () (const double Px,
15
                           const double Py,
                           const double Pz);
    virtual /*inline*/ Light & operator () (const char * str);
20
    virtual /*inline*/ Light & operator = (const int k);
    virtual /*inline*/ ostream & Ostream (ostream &) const;
friend istream & operator >> (istream & is, Light & light);
25
    virtual /*inline*/ Vector3D<double> Direction();
    virtual /*inline*/ Vector3D<double> Intensity();
    virtual /*inline*/ int Sharpness();

    virtual /*inline*/ Light & TransformDirection(const Matrix4x4<int> & M);
protected:
30
    Vector3D<double> direction;
    Vector3D<double> intensity;
    int sharpness;
};

35 #endif      // _Light_h_
:::::::cube4/LinearDataset.C
::::::: // LinearDataset.cpp
// (c) Ingmar Bitter '97
40 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

//">#define SHOW_ICON
//">#define DEBUG_SLC_READ
45
#include "LinearDataset.h"

void LinearDataset::Demo()
{
50

```

```

233
    LinearDataset           dataset("bulb.slc");
    cout << endl << "Demo of class " << typeid(dataset).name();
5     cout << endl << "size : " << sizeof(LinearDataset) << " Bytes";
     cout << endl << "public member functions:";
     cout << endl << "dataset.datasetFileName" << dataset.datasetFileName;
     cout << endl << "End of domo of class "<< typeid(dataset).name() << endl;
} // Demo

10
///////////
// constructors & destructors

15  LinearDataset::LinearDataset(const char * fileName)
{
    SetDatasetFile(fileName);
    ReadVolvisSLC();
} // constructor

20  LinearDataset::~LinearDataset()
{
    if (voxelData) delete voxelData;
} // destructor

25
///////////
// show/set data & data properties
//
// - local show/set functions

30  bool LinearDataset::SetDatasetFile(const char * fileName)
{
    // copy string starting with filename
    strcpy(datasetFileName, fileName);

35    // terminate filename at first space
    datasetFileName[strcspn(fileName, " ")] = 0;

    return strlen(datasetFileName) > 0;
} // SetDatasetFile

40
bool LinearDataset::ReadVolvisSLC()
{
    // load dataset temporarily into regular linear 3D grid array

45    cout << endl << "loading dataset \""<< datasetFileName << "\"";
    int magicDataTypeNumber;
    int u,v,w;
    int bitsPerVoxel;
    float unitLengthU, unitLengthV, unitLengthW;
50    int unitType, dataOrigin, dataModification, compressionType;
    unsigned char X;
}

```

55

```

5      ifstream srcFile(datasetFileName);

srcFile >> magicDataTypeNumber;
srcFile >> u >> v >> w; sizeUVW(u,v,w);
srcFile >> bitsPerVoxel;
srcFile >> unitLengthU >> unitLengthV >> unitLengthW;
srcFile >> unitType >> dataOrigin >> dataModification >> compressionType;

10     cout << endl << "magicDataTypeNumber:" << magicDataTypeNumber;
cout <<           " sizeUVW: " << sizeUVW;
cout <<           " bitsPerVoxel: " << bitsPerVoxel;
cout << endl << "unitLength: u=" << unitLengthU;
cout <<           " v=" << unitLengthV;
15     cout <<           " w=" << unitLengthW;
cout << endl << "unitType:" << TypeStr::Unit[unitType];
cout <<           " dataOrigin:" << TypeStr::DataOrigin[dataOrigin];
cout << endl << "dataModification:" <<
TypeStr::DataModification[dataModification];
20     cout <<           " compressionType:" <<
TypeStr::DataCompression[compressionType];

ReadVolvisIcon(srcFile);

// read dataset
25
int datasetSize    = sizeUVW.U() * sizeUVW.V() * sizeUVW.W();
int dataSliceSize = sizeUVW.U() * sizeUVW.V();
int compressedSize;

30     voxelData      = new unsigned char[datasetSize];
assert(voxelData);
unsigned char *compressedSlice = new unsigned char[2*dataSliceSize];
assert(compressedSlice);
unsigned char *slice = voxelData;

35     #ifdef DEBUG_SLC_READ
char str[500];
sprintf(str, " /bin/rm -f img/slcf*.miff & \n");
cout << str;
system(str);
#endif
40
#endif

45     cout << endl << "reading voxel data "; cout.flush();
for (int z=0; z<sizeUVW.W(); ++z, slice += dataSliceSize) {
    switch (compressionType) {
        case NO_COMPRESSION:           srcFile.read(slice,dataSliceSize);
break;
        case RUN_LENGTH_ENCODE:
            // read compressed slice size
            srcFile >> compressedSize;
            // advance to real data
50
}

```

```

235
for ( X = '\0'; X != 'X'; srcFile.get(X) ) ;

5      // read in slice
srcFile.read(compressedSlice, compressedSize);
assert( compressedSize == (int) srcFile.gcount() );

// decompress slice
// see also:
10     /home/fs2/mmshare/pkg/VolVis.2.1/lib/file_io/src/C_compression.c
register unsigned char *compressData = compressedSlice;
register unsigned char *voxel = slice;
register unsigned char currentValue = *compressData;
register unsigned char remaining;

15     for (currentValue = *(compressData++); (remaining =
(currentValue & 0x7f));
           currentValue = *(compressData++) .) {
20         if ( currentValue & 0x80 ) {
               while ( remaining-- )
                     *(voxel++) = *(compressData++);
         }
         else {
               currentValue = *(compressData++);
               while ( remaining-- )
                     *(voxel++) = currentValue;
25         }
         }
         break;
} // switch
if (z%10) cout << "..."; else cout << ":"; cerr.flush();

30 #ifdef DEBUG_SLC_READ
        // save slices as gray image
        char fileName[50];
        sprintf(fileName, "img/slc%03i.gray", z);
        (
5      ofstream slcGrayFile(fileName);
        slcGrayFile.write(slice,dataSliceSize); )
        sprintf(str,"convert -geometry 64x64 -size %ix%i img/slc%03i.gray
img/slc%03i.miff; /usr/bin/rm -f img/slc%03i.gray & \n",
               sizeUVW.U(), sizeUVW.V(),z,z,z);
        //cout << str;
        cout.flush();
40     system(str);
#endif
} // for z
delete compressedSlice; // free memory

45 #ifdef DEBUG_SLC_READ
        sprintf(str,"animate img/slc*.miff & \n");
        //cout << str;
        system(str);
#endif
50

```

236

```

    return true;
} // ReadVolvisSLC
5

void LinearDataset::ReadVolvisIcon(ifstream & srcFile)
{
    unsigned char *iconRed, *iconGreen, *iconBlue, X;
    int iconX, iconY;
10

    srcFile >> iconX >> iconY;

    cout << endl << "icon:" << iconX << "x" << iconY;

15    for ( X = '\0'; X != 'X'; srcFile.get(X) );

    // load icon
    iconRed = new unsigned char[iconX * iconY]; assert(iconRed);
    iconGreen = new unsigned char[iconX * iconY]; assert(iconGreen);
    iconBlue = new unsigned char[iconX * iconY]; assert(iconBlue);
20

    srcFile.read(iconRed, iconX*iconY);
    srcFile.read(iconGreen, iconX*iconY);
    srcFile.read(iconBlue, iconX*iconY);

25    #ifdef SHOW_ICON
        // save icon as seperate rgb file
        {
            ofstream iconFile("icon.rgb");
            iconFile.write(iconRed, iconX*iconY);
            iconFile.write(iconGreen, iconX*iconY);
            iconFile.write(iconBlue, iconX*iconY);
30        }
    #endif

        delete iconRed;
35        delete iconGreen;
        delete iconBlue;

    #ifdef SHOW_ICON
        // show icon
        char str[500];
40        sprintf(str,"convert -size %ix%i icon.rgb icon.miff ;",iconX,iconY);
        strcat(str," /bin/rm -f icon.rgb ; display -delay 10 icon.miff & \n");
        //cout << str;
        system(str);
    #endif
45    } // ReadVolvisIcon

    // end of LinearDataset.C
    ::::::::::::::::::::
    cube4/LinearDataset.h
    ::::::::::::::::::::
50    // LinearDataset.h

```

55

237

```

// (c) Ingmar Bitter '97

5   // Copyright, Mitsubishi Electric Information Technology Center
    // America, Inc., 1997, All rights reserved.

    #ifndef _LinearDataset_h_      // prevent multiple includes
    #define _LinearDataset_h_

10  #include <fstream.h>    // strcmp
    #include <iostream.h>   // strcmp
    #include <string.h>     // strcmp
    #include <strstream.h>  // strstream
    #include <stdio.h>       // sprintf
15  #include "Object.h"
    #include "Misc.h"
    #include "Vector3D.h"

20  class LinearDataset : virtual public Object {
public:

    static void      Demo ();

    // constructors & destructors
25  LinearDataset(const char * fileName);
    virtual ~LinearDataset();

    // show/set data & data properties

        // - local show/set functions
30  virtual      bool      ReadVolvisSLC();
    virtual      bool      SetDatasetFile(const char *);
public:
    unsigned char *voxelData;
    char datasetFileName[200];
    Vector3D<int> sizeUVW;

35  protected:
    virtual      void      ReadVolvisIcon(ifstream &);

};

40  #endif      // _LinearDataset_h_
:::::::::::
cube4/Makefile
:::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
45  #      make                  : compile only changed files and their depend
files
#      make all                : recompile all
#      make clean   : recompile all and remove unnecessary files
#
50  # rsh emerald "cd ~/ingmar/cube4 ; make"

```

55

238

```

# Copyright, Mitsubishi Electric Information Technology Center
5   # America, Inc., 1997, All rights reserved.

EXECUTABLE = go

SRC = main.C Test.C Object.C ModInt.C FixPointNumber.C Vector3D.C Matrix4x4.C \
10  Light.C \
      DynaArray.C Timer.C Misc.C FiFo.C Voxel.C Coxel.C Shadel.C
      LinearDataset.C Control.C AddressGenerator.C \
      VoxMem.C CoxMem.C MemoryCtrl.C \
      SliceVoxelFiFoStage.C SliceVoxelFiFoPipeline.C \
      TriLinZStage.C TriLinZPipeline.C \
      TriLinYStage.C TriLinYPipeline.C \
      TriLinXStage.C TriLinXPipeline.C \
      GradYStage.C GradYPipeline.C \
      GradXStage.C GradXPipeline.C \
      GradZStage.C GradZPipeline.C \
      GradZLinZStage.C GradZLinZPipeline.C \
      GradZLinYStage.C GradZLinYPipeline.C \
      GradZLinXStage.C GradZLinXPipeline.C \
      ColorLUT.C \
      ShaderStage.C ShaderPipeline.C ReflectanceMap.C \
      DataSyncStage.C DataSyncPipeline.C \
      ComposStage.C ComposPipeline.C \
      ComposBufferStage.C ComposBufferPipeline.C \
      ComposSelXStage.C ComposSelXPipeline.C \
      ComposSelYStage.C ComposSelYPipeline.C \
      ComposLinXStage.C ComposLinXPipeline.C \
      ComposLinYStage.C ComposLinYPipeline.C \
      FinalCoxelBuffer.C \
      Cube4.C

RM = /bin/rm -fs

35  #OPTIMIZE = -O3
    OPTIMIZE = -g
    WARN = -fullwarn -woff 1314
    CDEBUGFLAGS = $(OPTIMIZE) $(WARN)
    CCOPTIONS = -n32 -mips4 -r10000

40  CC = CC

INCLUDES =
#LIBS = -lfasttm -lm
LIBS = -lm
45  CFLAGS = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)
OFILES = $(SRC:.C=.o)

$(EXECUTABLE): $(OFILES)

50

```

55

```

239
$(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

5      #
#      target: dependency \n tab rule
#
.C.o:
        $(CC) $(CFLAGS) -c $<

10     #
#      switch:: ; \n tab rule
#
clean:      ;
        find . -name "**.o" -print -exec mv {} ~/dumpster \; ;
        find . -name "**~" -print -exec mv {} ~/dumpster \; ;
        find . -name "go" -print -exec mv {} ~/dumpster \; ;
        find . -name "core" -print -exec mv {} ~/dumpster \; ;

all:      ;
        pmake -u

20      depend:
        makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

25      main.o: Test.h Cube4.h /usr/include/string.h /usr/include/standards.h
main.o: /usr/include/stdcfa.h /usr/include/sgidefs.h /usr/include/stdio.h
main.o: /usr/include/assert.h Object.h Global.h Misc.h DynaArray.h
main.o: FixPointNumber.h Vector3D.h /usr/include/math.h ModInt.h
main.o: /usr/include/limits.h Matrix4x4.h FiFo.h Timer.h
main.o: /usr/include/sys/time.h /usr/include/sys/times.h
30      main.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h Shadel.h
main.o: Coxel.h Light.h LinearDataset.h VoxMem.h CoxMem.h Control.h
main.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
main.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
main.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
35      main.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
main.o: GradZStage.h GradZPipeline.h GradZLinZStage.h GradZLinZPipeline.h
main.o: GradZLinYStage.h GradZLinYPipeline.h GradZLinXStage.h
main.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h ReflectanceMap.h
main.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
main.o: ComposBufferPipeline.h ComposStage.h Compo$Pipeline.h
40      main.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
main.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
main.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
Test.o: Test.h Cube4.h /usr/include/string.h /usr/include/standards.h
Test.o: /usr/include/stdcfa.h /usr/include/sgidefs.h /usr/include/stdio.h
Test.o: /usr/include/assert.h Object.h Global.h Misc.h DynaArray.h
45      Test.o: FixPointNumber.h Vector3D.h /usr/include/math.h ModInt.h
Test.o: /usr/include/limits.h Matrix4x4.h FiFo.h Timer.h
Test.o: /usr/include/sys/time.h /usr/include/sys/times.h
Test.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h Shadel.h
Test.o: Coxel.h Light.h LinearDataset.h VoxMem.h CoxMem.h Control.h

```

50

55

```

240
Test.o: AddressGenerator.h MemoryCtrl1.h SliceVoxelFiFoStage.h
Test.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
5   Test.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
Test.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
Test.o: GradZStage.h GradZPipeline.h GradZLinZStage.h GradZLinZPipeline.h
Test.o: GradZLinYStage.h GradZLinYPipeline.h GradZLinXStage.h
Test.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h ReflectanceMap.h
10  Test.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
Test.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
15  Test.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
Test.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
Test.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h /usr/include/stdlib.h /usr/include/sgidefs.h
20  ModInt.o: ModInt.h Object.h /usr/include/string.h /usr/include/standards.h
ModInt.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
ModInt.o: /usr/include/sgidefs.h /usr/include/limits.h
FixPointNumber.o: FixPointNumber.h Object.h /usr/include/string.h
FixPointNumber.o: /usr/include/standards.h Global.h /usr/include/assert.h
25  FixPointNumber.o: /usr/include/stdlib.h /usr/include/sgidefs.h
Vector3D.o: Vector3D.h /usr/include/math.h /usr/include/sgidefs.h
Vector3D.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Vector3D.o: /usr/include/assert.h /usr/include/stdlib.h ModInt.h
Vector3D.o: /usr/include/limits.h
Matrix4x4.o: Matrix4x4.h /usr/include/math.h /usr/include/sgidefs.h
Matrix4x4.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
30  Matrix4x4.o: /usr/include/assert.h /usr/include/stdlib.h Vector3D.h ModInt.h
Matrix4x4.o: /usr/include/limits.h FixPointNumber.h
DynaArray.o: DynaArray.h /usr/include/assert.h Object.h /usr/include/string.h
DynaArray.o: /usr/include/standards.h Global.h /usr/include/stdlib.h
DynaArray.o: /usr/include/sgidefs.h
35  Timer.o: Timer.h /usr/include/sys/time.h /usr/include/standards.h
Timer.o: /usr/include/sgidefs.h /usr/include/sys/times.h
Timer.o: /usr/include/sys/types.h /usr/include/unistd.h Object.h
Timer.o: /usr/include/string.h Global.h /usr/include/assert.h
Timer.o: /usr/include/stdlib.h
Misc.o: Misc.h
40  Fifo.o: Fifo.h Object.h /usr/include/string.h /usr/include/standards.h
Fifo.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
Fifo.o: /usr/include/sgidefs.h
Voxel.o: Voxel.h Global.h /usr/include/assert.h /usr/include/stdlib.h
Voxel.o: /usr/include/standards.h /usr/include/sgidefs.h
45  Coxel.o: Coxel.h FixPointNumber.h Object.h /usr/include/string.h
Coxel.o: /usr/include/standards.h Global.h /usr/include/assert.h
Coxel.o: /usr/include/stdlib.h /usr/include/sgidefs.h
Shadel.o: Shadel.h FixPointNumber.h Object.h /usr/include/string.h
Shadel.o: /usr/include/standards.h Global.h /usr/include/assert.h
Shadel.o: /usr/include/stdlib.h /usr/include/sgidefs.h
50  Light.o: Light.h Object.h /usr/include/string.h /usr/include/standards.h
Light.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
Light.o: /usr/include/sgidefs.h Vector3D.h /usr/include/math.h ModInt.h
Light.o: /usr/include/limits.h Matrix4x4.h FixPointNumber.h
LinearDataset.o: LinearDataset.h /usr/include/string.h

```

241

```

LinearDataset.o:                               /usr/include/standards.h
/usr/include/stdio.h

5   LinearDataset.o: /usr/include/sgidefs.h Object.h Global.h
LinearDataset.o: /usr/include/assert.h /usr/include/stdlib.h Misc.h
LinearDataset.o: Vector3D.h /usr/include/math.h ModInt.h
LinearDataset.o: /usr/include/limits.h
Control.o: Control.h Misc.h Object.h /usr/include/string.h
Control.o: /usr/include/standards.h Global.h /usr/include/assert.h
10  Control.o: /usr/include/stdlib.h /usr/include/sgidefs.h ModInt.h
Control.o: /usr/include/limits.h FixPointNumber.h Vector3D.h
Control.o: /usr/include/math.h Matrix4x4.h
AddressGenerator.o: AddressGenerator.h Object.h /usr/include/string.h
AddressGenerator.o: /usr/include/standards.h Global.h /usr/include/assert.h
15  AddressGenerator.o: /usr/include/stdlib.h /usr/include/sgidefs.h Vector3D.h
AddressGenerator.o: /usr/include/math.h ModInt.h /usr/include/limits.h
AddressGenerator.o: Matrix4x4.h FixPointNumber.h Control.h Misc.h
VoxMem.o: VoxMem.h Object.h /usr/include/string.h /usr/include/standards.h
VoxMem.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
VoxMem.o: /usr/include/sgidefs.h Misc.h Voxel.h
20  CoxMem.o: CoxMem.h Object.h /usr/include/string.h /usr/include/standards.h
CoxMem.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
CoxMem.o: /usr/include/sgidefs.h Misc.h Coxel.h FixPointNumber.h
MemoryCtrl.o: MemoryCtrl.h Misc.h Object.h /usr/include/string.h
MemoryCtrl.o: /usr/include/standards.h Global.h /usr/include/assert.h
25  MemoryCtrl.o: /usr/include/stdlib.h /usr/include/sgidefs.h FixPointNumber.h
MemoryCtrl.o: Vector3D.h /usr/include/math.h ModInt.h /usr/include/limits.h
MemoryCtrl.o: Voxel.h VoxMem.h Control.h Matrix4x4.h
SliceVoxelFiFoStage.o: SliceVoxelFiFoStage.h Misc.h Object.h
SliceVoxelFiFoStage.o: /usr/include/string.h /usr/include/standards.h
30  SliceVoxelFiFoStage.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
SliceVoxelFiFoStage.o: /usr/include/sgidefs.h Voxel.h Control.h ModInt.h
SliceVoxelFiFoStage.o: /usr/include/limits.h FixPointNumber.h Vector3D.h
SliceVoxelFiFoStage.o: /usr/include/math.h Matrix4x4.h
SliceVoxelFiFoStage.o: SliceVoxelFiFoPipeline.h FiFo.h Coxel.h Cube4.h
SliceVoxelFiFoStage.o: /usr/include/stdio.h DynaArray.h Timer.h
35  SliceVoxelFiFoStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
SliceVoxelFiFoStage.o: /usr/include/sys/types.h /usr/include/unistd.h
SliceVoxelFiFoStage.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
SliceVoxelFiFoStage.o: AddressGenerator.h MemoryCtrl.h TriLinZStage.h
SliceVoxelFiFoStage.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
SliceVoxelFiFoStage.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
40  SliceVoxelFiFoStage.o: GradYPipeline.h GradXStage.h GradXPipeline.h
SliceVoxelFiFoStage.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
SliceVoxelFiFoStage.o: GradZLinZPipeline.h GradZLinYStage.h
SliceVoxelFiFoStage.o: GradZLinYPipeline.h GradZLinXStage.h
SliceVoxelFiFoStage.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
SliceVoxelFiFoStage.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
45  SliceVoxelFiFoStage.o: DataSyncPipeline.h ComposBufferStage.h
SliceVoxelFiFoStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
SliceVoxelFiFoStage.o: ComposSelXStage.h ComposSelXPipeline.h
SliceVoxelFiFoStage.o: ComposSelYStage.h ComposSelYPipeline.h
SliceVoxelFiFoStage.o: ComposLinXStage.h ComposLinXPipeline.h
SliceVoxelFiFoStage.o: ComposLinYStage.h ComposLinYPipeline.h
50

```

242

```

SliceVoxelFiFoStage.o: FinalCoxelBuffer.h
SliceVoxelFiFoPipeline.o: SliceVoxelFiFoPipeline.h Misc.h Object.h
5 SliceVoxelFiFoPipeline.o: /usr/include/string.h /usr/include/standards.h
SliceVoxelFiFoPipeline.o: Global.h /usr/include/assert.h
SliceVoxelFiFoPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h Fifo.h
SliceVoxelFiFoPipeline.o: Voxel.h Coxel.h FixPointNumber.h Control.h ModInt.h
SliceVoxelFiFoPipeline.o: /usr/include/limits.h Vector3D.h
SliceVoxelFiFoPipeline.o: /usr/include/math.h Matrix4x4.h
10 SliceVoxelFiFoPipeline.o: SliceVoxelFiFoStage.h Cube4.h /usr/include/stdio.h
SliceVoxelFiFoPipeline.o: DynaArray.h Timer.h /usr/include/sys/time.h
SliceVoxelFiFoPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
SliceVoxelFiFoPipeline.o: /usr/include/unistd.h Shadel.h Light.h
SliceVoxelFiFoPipeline.o: LinearDataset.h VoxMem.h CoxMem.h
15 SliceVoxelFiFoPipeline.o: AddressGenerator.h MemoryCtrl.h TriLinZStage.h
SliceVoxelFiFoPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
SliceVoxelFiFoPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
SliceVoxelFiFoPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
SliceVoxelFiFoPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
SliceVoxelFiFoPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
20 SliceVoxelFiFoPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
SliceVoxelFiFoPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
SliceVoxelFiFoPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
SliceVoxelFiFoPipeline.o: DataSyncPipeline.h ComposBufferStage.h
SliceVoxelFiFoPipeline.o: ComposBufferPipeline.h ComposStage.h
SliceVoxelFiFoPipeline.o: ComposPipeline.h ComposSelXStage.h
25 SliceVoxelFiFoPipeline.o: ComposSelXPipeline.h ComposSelYStage.h
SliceVoxelFiFoPipeline.o: ComposSelYPipeline.h ComposLinXStage.h
SliceVoxelFiFoPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
SliceVoxelFiFoPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinZStage.o: TriLinZStage.h Misc.h Object.h /usr/include/string.h
30 TriLinZStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
TriLinZStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h
TriLinZStage.o: Control.h ModInt.h /usr/include/limits.h FixPointNumber.h
TriLinZStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h TriLinZPipeline.h
TriLinZStage.o: Coxel.h Cube4.h /usr/include/stdio.h DynaArray.h Fifo.h
TriLinZStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
35 TriLinZStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
TriLinZStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
TriLinZStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
TriLinZStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
TriLinZStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
40 TriLinZStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h GradZLinZStage.h
TriLinZStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
TriLinZStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
TriLinZStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
TriLinZStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
TriLinZStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
45 TriLinZStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
TriLinZStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
TriLinZStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinZPipeline.o: TriLinZPipeline.h Misc.h Object.h /usr/include/string.h
TriLinZPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
TriLinZPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h

```

50

55

243

```

TriLinZPipeline.o: Coxel.h FixPointNumber.h Control.h ModInt.h
TriLinZPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
5 TriLinZPipeline.o: Matrix4x4.h TriLinZStage.h Cube4.h /usr/include/stdio.h
TriLinZPipeline.o: DynaArray.h FiFo.h Timer.h /usr/include/sys/time.h
TriLinZPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
TriLinZPipeline.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
TriLinZPipeline.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
TriLinZPipeline.o: SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
10 TriLinZPipeline.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
TriLinZPipeline.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
TriLinZPipeline.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
TriLinZPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
TriLinZPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
15 TriLinZPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
TriLinZPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
TriLinZPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
TriLinZPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
TriLinZPipeline.o: ComposSelYPipeline.h ComposLinXStage.h
20 TriLinZPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
TriLinZPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinYStage.o: TriLinYStage.h Misc.h Object.h /usr/include/string.h
TriLinYStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
TriLinYStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Vector3D.h
25 TriLinYStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Voxel.h
TriLinYStage.o: Control.h FixPointNumber.h Matrix4x4.h TriLinYPipeline.h
TriLinYStage.o: FiFo.h Coxel.h Cube4.h /usr/include/stdio.h DynaArray.h
TriLinYStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
TriLinYStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
30 TriLinYStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
TriLinYStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
TriLinYStage.o: TriLinZStage.h TriLinZPipeline.h TriLinXStage.h
TriLinYStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
TriLinYStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h GradZLinZStage.h
35 TriLinYStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
TriLinYStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
TriLinYStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
TriLinYStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
TriLinYStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
40 TriLinYStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
TriLinYStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
TriLinYStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinYPipeline.o: TriLinYPipeline.h Misc.h Object.h /usr/include/string.h
TriLinYPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
45 TriLinYPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h FiFo.h
TriLinYPipeline.o: Voxel.h Coxel.h FixPointNumber.h Control.h ModInt.h
TriLinYPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
TriLinYPipeline.o: Matrix4x4.h TriLinYStage.h Cube4.h /usr/include/stdio.h
TriLinYPipeline.o: DynaArray.h Timer.h /usr/include/sys/time.h
50 TriLinYPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
TriLinYPipeline.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
TriLinYPipeline.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
TriLinYPipeline.o: SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
TriLinYPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinXStage.h

```

244

```

5      TriLinYPipeline.o: TriLinXPipeline.h     GradYStage.h GradYPipeline.h
TriLinYPipeline.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
TriLinYPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
TriLinYPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
TriLinYPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
TriLinYPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
TriLinYPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
TriLinYPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
10     TriLinYPipeline.o: ComposSelYPipeline.h ComposLinXStage.h
TriLinYPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
TriLinYPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinXStage.o: TriLinXStage.h Misc.h Object.h /usr/include/string.h
TriLinXStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
15     TriLinXStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h
TriLinXStage.o: Control.h ModInt.h /usr/include/limits.h FixPointNumber.h
TriLinXStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h TriLinXPipeline.h
TriLinXStage.o: Coxel.h Cube4.h /usr/include/stdio.h DynaArray.h Fifo.h
TriLinXStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
TriLinXStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
20     TriLinXStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
TriLinXStage.o: MemoryCtrl.h SliceVoxelFifoStage.h SliceVoxelFifoPipeline.h
TriLinXStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
TriLinXStage.o: TriLinYPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
TriLinXStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h GradZLinZStage.h
TriLinXStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
25     TriLinXStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
TriLinXStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
TriLinXStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
TriLinXStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
TriLinXStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
30     TriLinXStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
TriLinXStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
TriLinXPipeline.o: TriLinXPipeline.h Misc.h Object.h /usr/include/string.h
TriLinXPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
TriLinXPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h
TriLinXPipeline.o: Coxel.h FixPointNumber.h Control.h ModInt.h
35     TriLinXPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
TriLinXPipeline.o: Matrix4x4.h TriLinXStage.h Cube4.h /usr/include/stdio.h
TriLinXPipeline.o: DynaArray.h Fifo.h Timer.h /usr/include/sys/time.h
TriLinXPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
TriLinXPipeline.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
TriLinXPipeline.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
40     TriLinXPipeline.o: SliceVoxelFifoStage.h SliceVoxelFifoPipeline.h
TriLinXPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
TriLinXPipeline.o: TriLinYPipeline.h GradYStage.h GradYPipeline.h
TriLinXPipeline.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
TriLinXPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
45     TriLinXPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
TriLinXPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
TriLinXPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
TriLinXPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
TriLinXPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
TriLinXPipeline.o: ComposSelYPipeline.h ComposLinXStage.h

```

50

55

245

```

TriLinXPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
TriLinXPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
5   GradyStage.o: GradyStage.h Misc.h Object.h /usr/include/string.h
GradyStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradyStage.o: /usr/include/stdcxx.h /usr/include/sgidefs.h Voxel.h Control.h
GradyStage.o: ModInt.h /usr/include/limits.h FixPointNumber.h Vector3D.h
GradyStage.o: /usr/include/math.h Matrix4x4.h GradyPipeline.h Fifo.h Coxel.h
10  GradyStage.o: Cube4.h /usr/include/stdio.h DynaArray.h Timer.h
GradyStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradyStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h Light.h
GradyStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
GradyStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
15  GradyStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
GradyStage.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h GraxStage.h
GradyStage.o: GraxPipeline.h GraxZStage.h GraxZPipeline.h GraxZLinZStage.h
GradyStage.o: GraxZLinZPipeline.h GraxZLinYStage.h GraxZLinYPipeline.h
GradyStage.o: GraxZLinXStage.h GraxZLinXPipeline.h ShaderStage.h
GradyStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
20  GradyStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
GradyStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
GradyStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
GradyStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
GradyStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
25  GradyPipeline.o: GradyPipeline.h Misc.h Object.h /usr/include/string.h
GradyPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradyPipeline.o: /usr/include/stdcxx.h /usr/include/sgidefs.h Fifo.h Voxel.h
GradyPipeline.o: Coxel.h FixPointNumber.h Control.h ModInt.h
GradyPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
GradyPipeline.o: Matrix4x4.h GradyStage.h Cube4.h /usr/include/stdio.h
30  GradyPipeline.o: DynaArray.h Timer.h /usr/include/sys/time.h
GradyPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
GradyPipeline.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
GradyPipeline.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
GradyPipeline.o: SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
GradyPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
35  GradyPipeline.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
GradyPipeline.o: GraxStage.h GraxXPipeline.h GraxZStage.h GraxZPipeline.h
GradyPipeline.o: GraxZStage.h GraxZLinZPipeline.h GraxZLinYStage.h
GradyPipeline.o: GraxZLinYPipeline.h GraxZLinXStage.h GraxZLinXPipeline.h
GradyPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
40  GradyPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
GradyPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradyPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradyPipeline.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradyPipeline.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
GraxStage.o: GraxStage.h Misc.h Object.h /usr/include/string.h
45  GraxStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
GraxStage.o: /usr/include/stdcxx.h /usr/include/sgidefs.h Voxel.h Control.h
GraxStage.o: ModInt.h /usr/include/limits.h FixPointNumber.h Vector3D.h
GraxStage.o: /usr/include/math.h Matrix4x4.h GraxXPipeline.h Fifo.h Coxel.h
GraxStage.o: Cube4.h /usr/include/stdio.h DynaArray.h Timer.h
GraxStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
50  GraxStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h Light.h

```

246

```

GradXStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
GradXStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
GradXStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
5 GradXStage.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h GradyStage.h
GradXStage.o: GradyPipeline.h GradZStage.h GradZPipeline.h GradZLinZStage.h
GradXStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
GradXStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
GradXStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
10 GradXStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
GradXStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
GradXStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
GradXStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
GradXStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
GradXPipeline.o: GradXPipeline.h Misc.h Object.h /usr/include/string.h
15 GradXPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradXPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h ModInt.h
GradXPipeline.o: /usr/include/limits.h Fifo.h Voxel.h Coxel.h
GradXPipeline.o: FixPointNumber.h Control.h Vector3D.h /usr/include/math.h
GradXPipeline.o: Matrix4x4.h GradXStage.h Cube4.h /usr/include/stdio.h
GradXPipeline.o: DynaArray.h Timer.h /usr/include/sys/time.h
20 GradXPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
GradXPipeline.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
GradXPipeline.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
GradXPipeline.o: SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
GradXPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
GradXPipeline.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
25 GradXPipeline.o: GradyStage.h GradyPipeline.h GradZStage.h GradZPipeline.h
GradXPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
GradXPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
GradXPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
GradXPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
30 GradXPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradXPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradXPipeline.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradXPipeline.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
GradZStage.o: GradZStage.h Misc.h Object.h /usr/include/string.h
GradZStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
35 GradZStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h Control.h
GradZStage.o: ModInt.h /usr/include/limits.h FixPointNumber.h Vector3D.h
GradZStage.o: /usr/include/math.h Matrix4x4.h GradZPipeline.h Coxel.h Cube4.h
GradZStage.o: /usr/include/stdio.h DynaArray.h Fifo.h Timer.h
GradZStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h Light.h
40 GradZStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
GradZStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
GradZStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
GradZStage.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h GradyStage.h
GradZStage.o: GradyPipeline.h GradXStage.h GradXPipeline.h GradZLinZStage.h
45 GradZStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
GradZStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
GradZStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
GradZStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
GradZStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h

```

50

55

247

```

GradZStage.o: ComposSelXPipeline.h           ComposSelYStage.h ComposSelYPipeline.h
GradZStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
GradZStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
5      GradZPipeline.o: GradZPipeline.h Misc.h Object.h /usr/include/string.h
GradZPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradZPipeline.o: /usr/include/stdcfa.h /usr/include/sgidefs.h Voxel.h Coxel.h
GradZPipeline.o: FixPointNumber.h Control.h ModInt.h /usr/include/limits.h
GradZPipeline.o: Vector3D.h /usr/include/math.h Matrix4x4.h GradZStage.h
10     GradZPipeline.o: Cube4.h /usr/include/stdio.h DynaArray.h FiFo.h Timer.h
GradZPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZPipeline.o: Light.h LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
GradZPipeline.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
GradZPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
15     GradZPipeline.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
GradZPipeline.o: GradyStage.h GradyPipeline.h GradXStage.h GradXPipeline.h
GradZPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
GradZPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
GradZPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
20     GradZPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
GradZPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradZPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradZPipeline.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradZPipeline.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
GradZLinZStage.o: GradZLinZStage.h Misc.h Object.h /usr/include/string.h
25     GradZLinZStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradZLinZStage.o: /usr/include/stdcfa.h /usr/include/sgidefs.h Vector3D.h
GradZLinZStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Voxel.h
GradZLinZStage.o: Control.h FixPointNumber.h Matrix4x4.h GradZLinZPipeline.h
GradZLinZStage.o: FiFo.h Coxel.h Cube4.h /usr/include/stdio.h DynaArray.h
GradZLinZStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
30     GradZLinZStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinZStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinZStage.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
GradZLinZStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
GradZLinZStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
GradZLinZStage.o: TriLinXPipeline.h GradyStage.h GradyPipeline.h GradXStage.h
35     GradZLinZStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h
GradZLinZStage.o: GradZLinYStage.h GradZLinYPipeline.h GradZLinXStage.h
GradZLinZStage.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
GradZLinZStage.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
GradZLinZStage.o: DataSyncPipeline.h ComposBufferStage.h
40     GradZLinZStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradZLinZStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradZLinZStage.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradZLinZStage.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
GradZLinZPipeline.o: GradZLinZPipeline.h Misc.h Object.h
GradZLinZPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
45     GradZLinZPipeline.o: /usr/include/assert.h /usr/include/stdcfa.h
GradZLinZPipeline.o: /usr/include/sgidefs.h FiFo.h Voxel.h Coxel.h
GradZLinZPipeline.o: FixPointNumber.h Control.h ModInt.h
GradZLinZPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
GradZLinZPipeline.o: Matrix4x4.h GradZLinZStage.h Cube4.h

```

50

55

248

```

GradZLinZPipeline.o:                               /usr/include/stdio.h DynaArray.h
Timer.h
5      GradZLinZPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZLinZPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinZPipeline.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinZPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
GradZLinZPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
GradZLinZPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinZPipeline.h
10     GradZLinZPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
GradZLinZPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
GradZLinZPipeline.o: GradZStage.h GradZPipeline.h GradZLinYStage.h
GradZLinZPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
GradZLinZPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h
15     GradZLinZPipeline.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h
GradZLinZPipeline.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
GradZLinZPipeline.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
GradZLinZPipeline.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
GradZLinZPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
GradZLinZPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
20     GradZLinYStage.o: GradZLinYStage.h Misc.h Object.h /usr/include/string.h
GradZLinYStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradZLinYStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Vector3D.h
GradZLinYStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Voxel.h
GradZLinYStage.o: Control.h FixPointNumber.h Matrix4x4.h GradZLinYPipeline.h
25     GradZLinYStage.o: FiFo.h Coxel.h Cube4.h /usr/include/stdio.h DynaArray.h
GradZLinYStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
GradZLinYStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinYStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinYStage.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
GradZLinYStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
30     GradZLinYStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
GradZLinYStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
GradZLinYStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h
GradZLinYStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinXStage.h
GradZLinYStage.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
GradZLinYStage.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
35     GradZLinYStage.o: DataSyncPipeline.h ComposBufferStage.h
GradZLinYStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradZLinYStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradZLinYStage.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradZLinYStage.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
40     GradZLinYPipeline.o: GradZLinYPipeline.h Misc.h Object.h
GradZLinYPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
GradZLinYPipeline.o: /usr/include/assert.h /usr/include/stdlib.h
GradZLinYPipeline.o: /usr/include/sgidefs.h FiFo.h Voxel.h Coxel.h
GradZLinYPipeline.o: FixPointNumber.h Control.h ModInt.h
GradZLinYPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
45     GradZLinYPipeline.o: Matrix4x4.h GradZLinYStage.h Cube4.h
GradZLinYPipeline.o: /usr/include/stdio.h DynaArray.h Timer.h
GradZLinYPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZLinYPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinYPipeline.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinYPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
50

```

249

```

GradZLinYPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
GradZLinYPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
GradZLinYPipeline.o: TriLinXStage.h TrilinXPipeline.h GradYStage.h
5 GradZLinYPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
GradZLinYPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
GradZLinYPipeline.o: GradZLinZPipeline.h GradZLinXStage.h GradZLinXPipeline.h
GradZLinYPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h
GradZLinYPipeline.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h
10 GradZLinYPipeline.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
GradZLinYPipeline.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
GradZLinYPipeline.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
GradZLinYPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
GradZLinYPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
GradZLinXStage.o: GradZLinXStage.h Misc.h Object.h /usr/include/string.h
15 GradZLinXStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
GradZLinXStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h
GradZLinXStage.o: Control.h ModInt.h /usr/include/limits.h FixPointNumber.h
GradZLinXStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h
GradZLinXStage.o: GradZLinXPipeline.h FiFo.h Coxel.h Cube4.h
GradZLinXStage.o: /usr/include/stdio.h DynaArray.h Timer.h
20 GradZLinXStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZLinXStage.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinXStage.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinXStage.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
GradZLinXStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
GradZLinXStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
25 GradZLinXStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
GradZLinXStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h
GradZLinXStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
GradZLinXStage.o: GradZLinYPipeline.h ShaderStage.h ShaderPipeline.h
GradZLinXStage.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
GradZLinXStage.o: DataSyncPipeline.h ComposBufferStage.h
30 GradZLinXStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
GradZLinXStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
GradZLinXStage.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
GradZLinXStage.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
GradZLinXPipeline.o: GradZLinXPipeline.h Misc.h Object.h
35 GradZLinXPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
GradZLinXPipeline.o: /usr/include/assert.h /usr/include/stdlib.h
GradZLinXPipeline.o: /usr/include/sgidefs.h FiFo.h Voxel.h Coxel.h
GradZLinXPipeline.o: FixPointNumber.h Control.h ModInt.h
GradZLinXPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
GradZLinXPipeline.o: Matrix4x4.h GradZLinXStage.h Cube4.h
GradZLinXPipeline.o: /usr/include/stdio.h DynaArray.h Timer.h
40 GradZLinXPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
GradZLinXPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Shadel.h
GradZLinXPipeline.o: Light.h LinearDataset.h VoxMem.h CoxMem.h
GradZLinXPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
GradZLinXPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
GradZLinXPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
45 GradZLinXPipeline.o: TriLinXStage.h TrilinXPipeline.h GradYStage.h
GradZLinXPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
GradZLinXPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h

```

50

55

250

```

GradZLinXPipeline.o:           GradZLinZPipeline.h GradZLinYStage.h
GradZLinYPipeline.h

5   GradZLinXPipeline.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h
GradZLinXPipeline.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h
GradZLinXPipeline.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
GradZLinXPipeline.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
GradZLinXPipeline.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
GradZLinXPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
10  GradZLinXPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ColorLUT.o: Cube4.h /usr/include/string.h /usr/include/standards.h
ColorLUT.o: /usr/include/stdlib.h /usr/include/sgidefs.h /usr/include/stdio.h
ColorLUT.o: /usr/include/assert.h Object.h Global.h Misc.h DynaArray.h
ColorLUT.o: FixPointNumber.h Vector3D.h /usr/include/math.h ModInt.h
15  ColorLUT.o: /usr/include/limits.h Matrix4x4.h Fifo.h Timer.h
ColorLUT.o: /usr/include/sys/time.h /usr/include/sys/times.h
ColorLUT.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h Shadel.h
ColorLUT.o: Coxel.h Light.h LinearDataset.h VoxMem.h CoxMem.h Control.h
ColorLUT.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFifoStage.h
ColorLUT.o: SliceVoxelFifoPipeline.h TriLinZStage.h TriLinZPipeline.h
20  ColorLUT.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
ColorLUT.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
ColorLUT.o: GradZStage.h GradZPipeline.h GradZLinZStage.h GradZLinZPipeline.h
ColorLUT.o: GradZLinYStage.h GradZLinYPipeline.h GradZLinXStage.h
ColorLUT.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
ColorLUT.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h DataSyncPipeline.h
25  ColorLUT.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
ColorLUT.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
ColorLUT.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
ColorLUT.o: ComposLinXPipeline.h ComposLinYStage.h ComposLinYPipeline.h
ColorLUT.o: FinalCoxelBuffer.h
30  ShaderStage.o: ShaderStage.h Misc.h Object.h /usr/include/string.h
ShaderStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
ShaderStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h Shadel.h
ShaderStage.o: FixPointNumber.h Control.h ModInt.h /usr/include/limits.h
ShaderStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h ShaderPipeline.h
ShaderStage.o: ReflectanceMap.h DynaArray.h Light.h ColorLUT.h Cube4.h
35  ShaderStage.o: /usr/include/stdio.h Fifo.h Timer.h /usr/include/sys/time.h
ShaderStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
ShaderStage.o: /usr/include/unistd.h Coxel.h LinearDataset.h VoxMem.h
ShaderStage.o: CoxMem.h AddressGenerator.h MemoryCtrl.h SliceVoxelFifoStage.h
ShaderStage.o: SliceVoxelFifoPipeline.h TriLinZStage.h TriLinZPipeline.h
40  ShaderStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
ShaderStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
ShaderStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h GradZLinZStage.h
ShaderStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
ShaderStage.o: GradZLinXStage.h GradZLinXPipeline.h DataSyncStage.h
ShaderStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
45  ShaderStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
ShaderStage.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
ShaderStage.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
ShaderStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ShaderPipeline.o: ShaderPipeline.h Misc.h Object.h /usr/include/string.h
ShaderPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h

```

50

55

251

```

ShaderPipeline.o: /usr/include/stdc-predef.h /usr/include/sgidefs.h Voxel.h
ShaderPipeline.o: Shadel.h FixPointNumber.h Control.h ModInt.h
5   ShaderPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
ShaderPipeline.o: Matrix4x4.h ReflectanceMap.h DynaArray.h Light.h ColorLUT.h
ShaderPipeline.o: ShaderStage.h Cube4.h /usr/include/stdio.h FiFo.h Timer.h
ShaderPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
ShaderPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Coxel.h
10  ShaderPipeline.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ShaderPipeline.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
ShaderPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
ShaderPipeline.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
ShaderPipeline.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
ShaderPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
15  ShaderPipeline.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
ShaderPipeline.o: GradZLinXStage.h GradZLinXPipeline.h DataSyncStage.h
ShaderPipeline.o: DataSyncPipeline.h ComposBufferStage.h
ShaderPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ShaderPipeline.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
20  ShaderPipeline.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
ShaderPipeline.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
ReflectanceMap.o: ReflectanceMap.h Misc.h Object.h /usr/include/string.h
ReflectanceMap.o: /usr/include/standards.h Global.h /usr/include/assert.h
ReflectanceMap.o: /usr/include/stdc-predef.h /usr/include/sgidefs.h
ReflectanceMap.o: FixPointNumber.h Vector3D.h /usr/include/math.h ModInt.h
25  ReflectanceMap.o: /usr/include/limits.h DynaArray.h Light.h Matrix4x4.h
ReflectanceMap.o: /usr/include/ctype.h /usr/include/stdio.h
DataSyncStage.o: DataSyncStage.h Misc.h Object.h /usr/include/string.h
DataSyncStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
DataSyncStage.o: /usr/include/stdc-predef.h /usr/include/sgidefs.h Vector3D.h
30  DataSyncStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Shadel.h
DataSyncStage.o: FixPointNumber.h Control.h Matrix4x4.h DataSyncPipeline.h
DataSyncStage.o: FiFo.h Cube4.h /usr/include/stdio.h DynaArray.h Timer.h
DataSyncStage.o: /usr/include/sys/time.h /usr/include/sys/times.h
DataSyncStage.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
DataSyncStage.o: Coxel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
35  DataSyncStage.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
DataSyncStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
DataSyncStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
DataSyncStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h GradXStage.h
DataSyncStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h
40  DataSyncStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
DataSyncStage.o: Shadel.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
DataSyncStage.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
DataSyncStage.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
DataSyncStage.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
45  DataSyncStage.o: ComposLinXPipeline.h ComposLinYStage.h ComposLinYPipeline.h
DataSyncStage.o: FinalCoxelBuffer.h
DataSyncPipeline.o: DataSyncPipeline.h Misc.h Object.h /usr/include/string.h
DataSyncPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
DataSyncPipeline.o: /usr/include/stdc-predef.h /usr/include/sgidefs.h FiFo.h
DataSyncPipeline.o: Shadel.h FixPointNumber.h Control.h ModInt.h
50  DataSyncPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h

```

252

```

DataSyncPipeline.o: Matrix4x4.h           DataSyncStage.h Cube4.h
/usr/include/stdio.h

5   DataSyncPipeline.o: DynaArray.h Timer.h /usr/include/sys/time.h
DataSyncPipeline.o: /usr/include/sys/times.h /usr/include/sys/types.h
DataSyncPipeline.o: /usr/include/unistd.h Voxel.h Coxel.h Light.h
DataSyncPipeline.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
DataSyncPipeline.o: MemoryCtrl.h SliceVoxelFiFoStage.h
DataSyncPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
10  DataSyncPipeline.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
DataSyncPipeline.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
DataSyncPipeline.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
DataSyncPipeline.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
DataSyncPipeline.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
DataSyncPipeline.o: ShaderStage.h ShaderPipeline.h ReflectMap.h
15  DataSyncPipeline.o: ColorLUT.h ComposBufferStage.h ComposBufferPipeline.h
DataSyncPipeline.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
DataSyncPipeline.o: ComposSelXPipeline.h ComposSelYStage.h
DataSyncPipeline.o: ComposSelYPipeline.h ComposLinXStage.h
DataSyncPipeline.o: ComposLinXPipeline.h ComposLinYStage.h
20  DataSyncPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposStage.o: ComposStage.h Misc.h Object.h /usr/include/string.h
ComposStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
ComposStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Voxel.h Coxel.h
ComposStage.o: FixPointNumber.h Control.h ModInt.h /usr/include/limits.h
ComposStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h ComposPipeline.h
25  ComposStage.o: Shadel.h Cube4.h /usr/include/stdio.h DynaArray.h Fifo.h
ComposStage.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
ComposStage.o: /usr/include/sys/types.h /usr/include/unistd.h Light.h
ComposStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
ComposStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
30  ComposStage.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
ComposStage.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
ComposStage.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
ComposStage.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
ComposStage.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
35  ComposStage.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h DataSyncStage.h
ComposStage.o: DataSyncPipeline.h ComposBufferStage.h ComposBufferPipeline.h
ComposStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
ComposStage.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
ComposStage.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
ComposPipeline.o: ComposPipeline.h Misc.h Object.h /usr/include/string.h
40  ComposPipeline.o: /usr/include/standards.h Global.h /usr/include/assert.h
ComposPipeline.o: /usr/include/stdlib.h /usr/include/sgidefs.h Shadel.h
ComposPipeline.o: FixPointNumber.h Coxel.h Control.h ModInt.h
ComposPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
ComposPipeline.o: Matrix4x4.h ComposStage.h Voxel.h Cube4.h
ComposPipeline.o: /usr/include/stdio.h DynaArray.h Fifo.h Timer.h
45  ComposPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
ComposPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Light.h
ComposPipeline.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposPipeline.o: MemoryCtrl.h SliceVoxelFiFoStage.h SliceVoxelFiFoPipeline.h
ComposPipeline.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h

```

50

55

253

```

ComposPipeline.o: TriLinYPipeline.h      TriLinXStage.h TriLinXPipeline.h
ComposPipeline.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
ComposPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
5      ComposPipeline.o: GradZLinZPipeline.h GradZLinYStage.h GradZLinYPipeline.h
ComposPipeline.o: GradZLinXStage.h GradZLinXPipeline.h ShaderStage.h
ComposPipeline.o: ShaderPipeline.h ReflectanceMap.h ColorLUT.h
ComposPipeline.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
ComposPipeline.o: ComposBufferPipeline.h ComposSelXStage.h
10     ComposPipeline.o: ComposSelXPipeline.h ComposSelYStage.h ComposSelYPipeline.h
ComposPipeline.o: ComposLinXStage.h ComposLinXPipeline.h ComposLinYStage.h
ComposPipeline.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposBufferStage.o: ComposBufferStage.h Misc.h Object.h
ComposBufferStage.o: /usr/include/string.h /usr/include/standards.h Global.h
15     ComposBufferStage.o: /usr/include/assert.h /usr/include/stdlib.h
ComposBufferStage.o: /usr/include/sgidefs.h Voxel.h Coxel.h FixPointNumber.h
ComposBufferStage.o: ComposBufferPipeline.h Control.h ModInt.h
ComposBufferStage.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
ComposBufferStage.o: Matrix4x4.h Fifo.h Cube4.h /usr/include/stdio.h
ComposBufferStage.o: DynaArray.h Timer.h /usr/include/sys/time.h
20     ComposBufferStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
ComposBufferStage.o: /usr/include/unistd.h Shadel.h Light.h LinearDataset.h
ComposBufferStage.o: VoxMem.h CoxMem.h AddressGenerator.h MemoryCtrl.h
ComposBufferStage.o: SliceVoxelFifoStage.h SliceVoxelFifoPipeline.h
ComposBufferStage.o: TriLinZStage.h TriLinZPipeline.h TriLinYStage.h
ComposBufferStage.o: TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
25     ComposBufferStage.o: GradYStage.h GradYPipeline.h GradXStage.h
ComposBufferStage.o: GradXPipeline.h GradZStage.h GradZPipeline.h
ComposBufferStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
ComposBufferStage.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
ComposBufferStage.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h
ComposBufferStage.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h
30     ComposBufferStage.o: ComposStage.h ComposPipeline.h ComposSelXStage.h
ComposBufferStage.o: ComposSelXPipeline.h ComposSelYStage.h
ComposBufferStage.o: ComposSelYPipeline.h ComposLinXStage.h
ComposBufferStage.o: ComposLinXPipeline.h ComposLinYStage.h
ComposBufferStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposBufferPipeline.o: ComposBufferPipeline.h Misc.h Object.h
35     ComposBufferPipeline.o: /usr/include/string.h /usr/include/standards.h
ComposBufferPipeline.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
ComposBufferPipeline.o: /usr/include/sgidefs.h Voxel.h Coxel.h
ComposBufferPipeline.o: FixPointNumber.h Control.h ModInt.h
ComposBufferPipeline.o: /usr/include/limits.h Vector3D.h /usr/include/math.h
ComposBufferPipeline.o: Matrix4x4.h Fifo.h ComposBufferStage.h Cube4.h
40     ComposBufferPipeline.o: /usr/include/stdio.h DynaArray.h Timer.h
ComposBufferPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
ComposBufferPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h
ComposBufferPipeline.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
ComposBufferPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFifoStage.h
45     ComposBufferPipeline.o: SliceVoxelFifoPipeline.h TriLinZStage.h
ComposBufferPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
ComposBufferPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
ComposBufferPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
ComposBufferPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h

```

50

55

254

```

ComposBufferPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
ComposBufferPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
ComposBufferPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
ComposBufferPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
ComposBufferPipeline.o: DataSyncPipeline.h ComposStage.h ComposPipeline.h
ComposBufferPipeline.o: ComposSelXStage.h ComposSelXPipeline.h
ComposBufferPipeline.o: ComposSelYStage.h ComposSelYPipeline.h
ComposBufferPipeline.o: ComposLinXStage.h ComposLinXPipeline.h
ComposBufferPipeline.o: ComposLinYStage.h ComposLinYPipeline.h
ComposBufferPipeline.o: FinalCoxelBuffer.h
ComposSelXStage.o: ComposSelXStage.h Misc.h Object.h /usr/include/string.h
ComposSelXStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
ComposSelXStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Coxel.h
ComposSelXStage.o: FixPointNumber.h Control.h ModInt.h /usr/include/limits.h
ComposSelXStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h
ComposSelXStage.o: ComposSelXPipeline.h Cube4.h /usr/include/stdio.h
ComposSelXStage.o: DynaArray.h FiFo.h Timer.h /usr/include/sys/time.h
ComposSelXStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
ComposSelXStage.o: /usr/include/unistd.h Voxel.h Shadel.h Light.h
ComposSelXStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposSelXStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h
ComposSelXStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
ComposSelXStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
ComposSelXStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
ComposSelXStage.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
ComposSelXStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
ComposSelXStage.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
ComposSelXStage.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
ComposSelXStage.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
ComposSelXStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposSelXStage.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
ComposSelXStage.o: ComposLinXPipeline.h ComposLinYStage.h
ComposSelXStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposSelXPipeline.o: ComposSelXPipeline.h Misc.h Object.h
ComposSelXPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
ComposSelXPipeline.o: /usr/include/assert.h /usr/include/stdlib.h
ComposSelXPipeline.o: /usr/include/sgidefs.h ModInt.h /usr/include/limits.h
ComposSelXPipeline.o: Coxel.h FixPointNumber.h Control.h Vector3D.h
ComposSelXPipeline.o: /usr/include/math.h Matrix4x4.h ComposSelXStage.h
ComposSelXPipeline.o: Cube4.h /usr/include/stdio.h DynaArray.h FiFo.h Timer.h
ComposSelXPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
ComposSelXPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
ComposSelXPipeline.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
ComposSelXPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
ComposSelXPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
ComposSelXPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
ComposSelXPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
ComposSelXPipeline.o: GradXPipeline.h GradXStage.h GradXPipeline.h
ComposSelXPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
ComposSelXPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
ComposSelXPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
ComposSelXPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
ComposSelXPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h

```

50

55

255

```

ComposSelXPipeline.o: DataSyncPipeline.h ComposBufferStage.h
ComposSelXPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposSelXPipeline.o: ComposSelYStage.h ComposSelYPipeline.h
5 ComposSelXPipeline.o: ComposLinXStage.h ComposLinXPipeline.h
ComposSelXPipeline.o: ComposLinYStage.h ComposLinYPipeline.h
ComposSelXPipeline.o: FinalCoxelBuffer.h
ComposSelYStage.o: ComposSelYStage.h Misc.h Object.h /usr/include/string.h
ComposSelYStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
10 ComposSelYStage.o: /usr/include/stdlib.h /usr/include/sgidefs.h Vector3D.h
ComposSelYStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Coxel.h
ComposSelYStage.o: FixPointNumber.h Control.h Matrix4x4.h
ComposSelYStage.o: ComposSelYPipeline.h FiFo.h Cube4.h /usr/include/stdio.h
ComposSelYStage.o: DynaArray.h Timer.h /usr/include/sys/time.h
ComposSelYStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
15 ComposSelYStage.o: /usr/include/unistd.h Voxel.h Shadel.h Light.h
ComposSelYStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposSelYStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h
ComposSelYStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
ComposSelYStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
20 ComposSelYStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
ComposSelYStage.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
ComposSelYStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
ComposSelYStage.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
ComposSelYStage.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
ComposSelYStage.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
25 ComposSelYStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposSelYStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposLinXStage.h
ComposSelYStage.o: ComposLinXPipeline.h ComposLinYStage.h
ComposSelYStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposSelYPipeline.o: ComposSelYPipeline.h Misc.h Object.h
ComposSelYPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
30 ComposSelYPipeline.o: /usr/include/assert.h /usr/include/stdlib.h
ComposSelYPipeline.o: /usr/include/sgidefs.h FiFo.h Coxel.h FixPointNumber.h
ComposSelYPipeline.o: Control.h ModInt.h /usr/include/limits.h Vector3D.h
ComposSelYPipeline.o: /usr/include/math.h Matrix4x4.h ComposSelYStage.h
ComposSelYPipeline.o: Cube4.h /usr/include/stdio.h DynaArray.h Timer.h
ComposSelYPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
35 ComposSelYPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
ComposSelYPipeline.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
ComposSelYPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
ComposSelYPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
ComposSelYPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
ComposSelYPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
40 ComposSelYPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
ComposSelYPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
ComposSelYPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
ComposSelYPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
ComposSelYPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
ComposSelYPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
45 ComposSelYPipeline.o: DataSyncPipeline.h ComposBufferStage.h
ComposSelYPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposSelYPipeline.o: ComposSelXStage.h ComposSelXPipeline.h
ComposSelYPipeline.o: ComposLinXStage.h ComposLinXPipeline.h

```

50

55

256

```

ComposSelYPipeline.o: ComposLinYStage.h ComposLinYPipeline.h
ComposSelYPipeline.o: FinalCoxelBuffer.h
5 ComposLinXStage.o: ComposLinXStage.h Misc.h Object.h /usr/include/string.h
ComposLinXStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
ComposLinXStage.o: /usr/include/stdc硐.h /usr/include/sgidefs.h Coxel.h
ComposLinXStage.o: FixPointNumber.h Control.h ModInt.h /usr/include/limits.h
ComposLinXStage.o: Vector3D.h /usr/include/math.h Matrix4x4.h
ComposLinXStage.o: ComposLinXPipeline.h Cube4.h /usr/include/stdio.h
10 ComposLinXStage.o: DynaArray.h FiFo.h Timer.h /usr/include/sys/time.h
ComposLinXStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
ComposLinXStage.o: /usr/include/unistd.h Voxel.h Shadel.h Light.h
ComposLinXStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposLinXStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h
15 ComposLinXStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
ComposLinXStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
ComposLinXStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
ComposLinXStage.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
ComposLinXStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
20 ComposLinXStage.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
ComposLinXStage.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
ComposLinXStage.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
ComposLinXStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposLinXStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
ComposLinXStage.o: ComposSelYPipeline.h ComposLinYStage.h
25 ComposLinXStage.o: ComposLinYPipeline.h FinalCoxelBuffer.h
ComposLinXPipeline.o: ComposLinXPipeline.h Misc.h Object.h
ComposLinXPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
ComposLinXPipeline.o: /usr/include/assert.h /usr/include/stdc硐.h
ComposLinXPipeline.o: /usr/include/sgidefs.h Coxel.h FixPointNumber.h
ComposLinXPipeline.o: Control.h ModInt.h /usr/include/limits.h Vector3D.h
30 ComposLinXPipeline.o: /usr/include/math.h Matrix4x4.h ComposLinXStage.h
ComposLinXPipeline.o: Cube4.h /usr/include/stdio.h DynaArray.h FiFo.h Timer.h
ComposLinXPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
ComposLinXPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
ComposLinXPipeline.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
ComposLinXPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
35 ComposLinXPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
ComposLinXPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
ComposLinXPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
ComposLinXPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
ComposLinXPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
ComposLinXPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
40 ComposLinXPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
ComposLinXPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
ComposLinXPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
ComposLinXPipeline.o: DataSyncPipeline.h ComposBufferStage.h
ComposLinXPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
45 ComposLinXPipeline.o: ComposSelXStage.h ComposSelXPipeline.h
ComposLinXPipeline.o: ComposSelYStage.h ComposSelYPipeline.h
ComposLinXPipeline.o: ComposLinYStage.h ComposLinYPipeline.h
ComposLinXPipeline.o: FinalCoxelBuffer.h
ComposLinYStage.o: ComposLinYStage.h Misc.h Object.h /usr/include/string.h
ComposLinYStage.o: /usr/include/standards.h Global.h /usr/include/assert.h
50

```

55

257

```

ComposLinYStage.o:                               /usr/include/stdlib.h
/usr/include/sgidefs.h Vector3D.h
5      ComposLinYStage.o: /usr/include/math.h ModInt.h /usr/include/limits.h Coxel.h
ComposLinYStage.o: FixPointNumber.h Control.h Matrix4x4.h
ComposLinYStage.o: ComposLinYPipeline.h FiFo.h Cube4.h /usr/include/stdio.h
ComposLinYStage.o: DynaArray.h Timer.h /usr/include/sys/time.h
ComposLinYStage.o: /usr/include/sys/times.h /usr/include/sys/types.h
ComposLinYStage.o: /usr/include/unistd.h Voxel.h Shadel.h Light.h
10     ComposLinYStage.o: LinearDataset.h VoxMem.h CoxMem.h AddressGenerator.h
ComposLinYStage.o: MemoryCtrl.h SliceVoxelFiFoStage.h
ComposLinYStage.o: SliceVoxelFiFoPipeline.h TriLinZStage.h TriLinZPipeline.h
ComposLinYStage.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
ComposLinYStage.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
15     ComposLinYStage.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
ComposLinYStage.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
ComposLinYStage.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
ComposLinYStage.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h ColorLUT.h
ComposLinYStage.o: DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
ComposLinYStage.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
20     ComposLinYStage.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
ComposLinYStage.o: ComposSelYPipeline.h ComposLinXStage.h
ComposLinYStage.o: ComposLinXPipeline.h FinalCoxelBuffer.h
ComposLinYPipeline.o: ComposLinYPipeline.h Misc.h Object.h
ComposLinYPipeline.o: /usr/include/string.h /usr/include/standards.h Global.h
ComposLinYPipeline.o: /usr/include/assert.h /usr/include/stdlib.h
25     ComposLinYPipeline.o: /usr/include/sgidefs.h FiFo.h Coxel.h FixPointNumber.h
ComposLinYPipeline.o: Control.h ModInt.h /usr/include/limits.h Vector3D.h
ComposLinYPipeline.o: /usr/include/math.h Matrix4x4.h ComposLinYStage.h
ComposLinYPipeline.o: Cube4.h /usr/include/stdio.h DynaArray.h Timer.h
ComposLinYPipeline.o: /usr/include/sys/time.h /usr/include/sys/times.h
30     ComposLinYPipeline.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
ComposLinYPipeline.o: Shadel.h Light.h LinearDataset.h VoxMem.h CoxMem.h
ComposLinYPipeline.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFiFoStage.h
ComposLinYPipeline.o: SliceVoxelFiFoPipeline.h TriLinZStage.h
ComposLinYPipeline.o: TriLinZPipeline.h TriLinYStage.h TriLinYPipeline.h
ComposLinYPipeline.o: TriLinXStage.h TriLinXPipeline.h GradYStage.h
35     ComposLinYPipeline.o: GradYPipeline.h GradXStage.h GradXPipeline.h
ComposLinYPipeline.o: GradZStage.h GradZPipeline.h GradZLinZStage.h
ComposLinYPipeline.o: GradZLinZPipeline.h GradZLinYStage.h
ComposLinYPipeline.o: GradZLinYPipeline.h GradZLinXStage.h
ComposLinYPipeline.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h
40     ComposLinYPipeline.o: ReflectanceMap.h ColorLUT.h DataSyncStage.h
ComposLinYPipeline.o: DataSyncPipeline.h ComposBufferStage.h
ComposLinYPipeline.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
ComposLinYPipeline.o: ComposSelXStage.h ComposSelXPipeline.h
ComposLinYPipeline.o: ComposSelYStage.h ComposSelYPipeline.h
ComposLinYPipeline.o: ComposLinXStage.h ComposLinXPipeline.h
45     ComposLinYPipeline.o: FinalCoxelBuffer.h
FinalCoxelBuffer.o: FinalCoxelBuffer.h Object.h /usr/include/string.h
FinalCoxelBuffer.o: /usr/include/standards.h Global.h /usr/include/assert.h
FinalCoxelBuffer.o: /usr/include/stdlib.h /usr/include/sgidefs.h Control.h
FinalCoxelBuffer.o: Misc.h ModInt.h /usr/include/limits.h FixPointNumber.h
FinalCoxelBuffer.o: Vector3D.h /usr/include/math.h Matrix4x4.h Coxel.h
50

```

55

258

```

FinalCoxelBuffer.o: CoxMem.h Cube4.h      /usr/include/stdio.h DynaArray.h Fifo.h
FinalCoxelBuffer.o: Timer.h /usr/include/sys/time.h /usr/include/sys/times.h
FinalCoxelBuffer.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h
FinalCoxelBuffer.o: Shadel.h Light.h LinearDataset.h VoxMem.h
FinalCoxelBuffer.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFifoStage.h
FinalCoxelBuffer.o: SliceVoxelFifoPipeline.h TriLinZStage.h TriLinZPipeline.h
FinalCoxelBuffer.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h
FinalCoxelBuffer.o: TriLinXPipeline.h GradYStage.h GradYPipeline.h
FinalCoxelBuffer.o: GradXStage.h GradXPipeline.h GradZStage.h GradZPipeline.h
FinalCoxelBuffer.o: GradZLinZStage.h GradZLinZPipeline.h GradZLinYStage.h
FinalCoxelBuffer.o: GradZLinYPipeline.h GradZLinXStage.h GradZLinXPipeline.h
FinalCoxelBuffer.o: ShaderStage.h ShaderPipeline.h ReflectanceMap.h
FinalCoxelBuffer.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h
FinalCoxelBuffer.o: ComposBufferStage.h ComposBufferPipeline.h ComposStage.h
FinalCoxelBuffer.o: ComposPipeline.h ComposSelXStage.h ComposSelXPipeline.h
FinalCoxelBuffer.o: ComposSelYStage.h ComposSelYPipeline.h ComposLinXStage.h
FinalCoxelBuffer.o: ComposLinXPipeline.h ComposLinYStage.h
FinalCoxelBuffer.o: ComposLinYPipeline.h
Cube4.o: Cube4.h /usr/include/string.h /usr/include/standards.h
Cube4.o: /usr/include/stdlib.h /usr/include/sgidefs.h /usr/include/stdio.h
Cube4.o: /usr/include/assert.h Object.h Global.h Misc.h DynaArray.h
Cube4.o: FixPointNumber.h Vector3D.h /usr/include/math.h ModInt.h
Cube4.o: /usr/include/limits.h Matrix4x4.h Fifo.h Timer.h
Cube4.o: /usr/include/sys/time.h /usr/include/sys/times.h
Cube4.o: /usr/include/sys/types.h /usr/include/unistd.h Voxel.h Shadel.h
Cube4.o: Coxel.h Light.h LinearDataset.h VoxMem.h CoxMem.h Control.h
Cube4.o: AddressGenerator.h MemoryCtrl.h SliceVoxelFifoStage.h
Cube4.o: SliceVoxelFifoPipeline.h TriLinZStage.h TriLinZPipeline.h
Cube4.o: TriLinYStage.h TriLinYPipeline.h TriLinXStage.h TriLinXPipeline.h
Cube4.o: GradYStage.h GradYPipeline.h GradXStage.h GradXPipeline.h
Cube4.o: GradZStage.h GradZPipeline.h GradZLinZStage.h GradZLinZPipeline.h
Cube4.o: GradZLinYStage.h GradZLinYPipeline.h GradZLinXStage.h
Cube4.o: GradZLinXPipeline.h ShaderStage.h ShaderPipeline.h ReflectanceMap.h
Cube4.o: ColorLUT.h DataSyncStage.h DataSyncPipeline.h ComposBufferStage.h
Cube4.o: ComposBufferPipeline.h ComposStage.h ComposPipeline.h
Cube4.o: ComposSelXStage.h ComposSelXPipeline.h ComposSelYStage.h
Cube4.o: ComposSelYPipeline.h ComposLinXStage.h ComposLinXPipeline.h
Cube4.o: ComposLinYStage.h ComposLinYPipeline.h FinalCoxelBuffer.h
Cube4.o: Cube4Interface.C Cube4Debug.C
:::::::::::
cube4/MemoryCtrl.C
:::::::::::
// MemoryCtrl.C
// (c) Ingmar Bitter '97
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "MemoryCtrl.h"
void MemoryCtrl::Demo()
{
    MemoryCtrl control;

```

50

55

```

259
cout << endl << "Demo of class " << typeid(control).name();
cout << endl << "size : " << sizeof(MemoryCtrl) << " Bytes";
cout << endl << "public member functions:";
cout << endl << "MemoryCtrl control; = " << control;
cout << endl << "End of demo of class " << typeid(control).name() << endl;
} // Demo

10   /////////////////////////////////
// constructors & destructors

// static first init
int MemoryCtrl::numOfChips           = 0;
int MemoryCtrl::numOfPipelinesPerChip = 0;
15

MemoryCtrl::MemoryCtrl()
{
    results.voxel = new Voxel [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
[ numOfPipelinesPerChip ];
} // constructor

MemoryCtrl::~MemoryCtrl()
25
{
    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
30
    }
} // destructor

35   /////////////////////////////////
// show/set data & data properties

ostream & MemoryCtrl::Ostream(ostream & os) const
{
    // append MemoryCtrl info to os
40    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex            = " << chipIndex;
    os << endl << "  memory                = " << mem;
45
    // return complete os
    return os;
}

50   ) // Ostream

```

55

```

5   //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

10  void MemoryCtrl::GlobalSetup(const int setNumOfChips,
{
    const int setNumOfPipelinesPerChip)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
} // GlobalSetup

15

void MemoryCtrl::LocalSetup(const int setChipIndex, VoxMem * memory)
{
    chipIndex      = setChipIndex;
    mem = memory;
} // LocalSetup

20

void MemoryCtrl::PerFrameSetup()
{
    // reset pipeline registers
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p] = 0;
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // Init
25

30

35

40

45

50
//////////////////////////////////////////////////////////////////
// local computation functions

void MemoryCtrl::RunForOneClockCycle()
{
    // add0 = cube4->addGen[0].MemoryAddress(x ,y,z);
    // add1 = cube4->addGen[0].MemoryAddress(x+1,y,z);

    // for startOfVolume timing in SliceVoxelFIFO
    results.delayedPerChipControlFlags = results.perChipControlFlags;

    // normal stuff do do at each clock cycle
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p] = (*mem)(inputs.address[p]);
        results.weightsXYZ[p] = inputs.weightsXYZ[p];
}

```

```

      results.perPipelineControlFlags[p] = inputs.perPipelineControlFlags[p];
    }
    results.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle

10   /////////////////////////////////
// internal utility functions

// end of MemoryCtrl.C
:::::::::::
15   cube4/MemoryCtrl.h
:::::::::::
// MemoryCtrl.h
// (c) Ingmar Bitter '97

20   // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _MemoryCtrl_h_ // prevent multiple includes
#define _MemoryCtrl_h_

25   #include "Misc.h"
#include "Object.h"
#include "FixPointNumber.h"
#include "Vector3D.h"
#include "Voxel.h"
#include "VoxMem.h"
30   #include "Control.h"

        class MemoryCtrlInputs {
public: // pointers
        int             *address;
35       Vector3D<FixPointNumber> *weightsXYZ;
        PerChipControlFlags     *perChipControlFlags;
        PerPipelineControlFlags *perPipelineControlFlags;
    };

40       class MemoryCtrlResults {
public: // arrays
        Voxel           *voxel;
        Vector3D<FixPointNumber> *weightsXYZ;
        PerChipControlFlags     perChipControlFlags;
45       PerChipControlFlags     delayedPerChipControlFlags;
        PerPipelineControlFlags *perPipelineControlFlags;
    };

50       class MemoryCtrl : virtual public Object {
public:

```

```

5      static void      Demo ();
10     // constructors & destructors
11     MemoryCtrl ();
12     ~MemoryCtrl ();
13
14     // show/set data & data properties
15     // - class Object requirements
16     virtual ostream & Ostream (ostream & ) const;
17
18     // - local show/set functions
19     static void GlobalSetup (const int setNumOfChips,
20
21             const int setNumOfPipelinesPerChip);
22
23     virtual void LocalSetup(const int setChipIndex, VoxMem * memory);
24     virtual void PerFrameSetup();
25
26     // local computation functions
27     virtual void RunForOneClockCycle();
28
29     public:
30         MemoryCtrlInputs inputs;
31         MemoryCtrlResults results;
32
33     protected:
34         VoxMem *mem;
35         static int numOfChips, numOfPipelinesPerChip;
36         int          chipIndex;
37         // static Cube4 *cube4;
38     };
39
40     #endif      // _MemoryCtrl_h_
41     ::::::::::::::
42     cube4/Misc.C
43     ::::::::::::::
44     // Misc.C
45     // (c) Ingmar Bitter '97 / Urs Kanus '97
46
47     // Copyright, Mitsubishi Electric Information Technology Center
48     // America, Inc., 1997, All rights reserved.
49
50     #include "Misc.h"
51
52     // Cube4 stuff
53     const char * TypeStr::ProjectionStyle[] = PROJECTION_STYLES_STR;
54     const char * TypeStr::CompositingStyle[] = COMPOSITING_STYLES_STR;
55     const char * TypeStr::ShaderMode[] = SHADER_MODES_STR;
56     const char * TypeStr::CubeMode[] = CUBE_MODES_STR;
57     const char * TypeStr::MajorViewDirection[] = MAJOR_VIEW_DIRECTION_STR;
58     const char * TypeStr::XStepDirection[] = X_STEP_DIRECTION_STR;
59     const char * TypeStr::YStepDirection[] = Y_STEP_DIRECTION_STR;
60     const char * TypeStr::RayEndFace[] = RAY_END_FACES_STR;

```

```

// VolVis SLC stuff
5   const char * TypeStr::Unit[]           = UNIT_TYPE_STR;
    const char * TypeStr::DataOrigin[]      = DATA_ORIGIN_TYPE_STR;
    const char * TypeStr::DataModification[] = DATA_MODIFICATION_TYPE_STR;
    const char * TypeStr::DataCompression[] = DATA_COMPRESSION_TYPE_STR;

// end of Misc.C
10  ::::::::::::::::::::
    cube4/Misc.h
    ::::::::::::::::::::
    // Misc.h
    // (c) Ingmar Bitter '97

15  // Copyright, Mitsubishi Electric Information Technology Center
    // America, Inc., 1997, All rights reserved.

#ifndef _Misc_h_ // prevent multiple includes
#define _Misc_h_
20  typedef int ScalarGradient;

typedef enum { parallel, perspective } ProjectionStyles;
#define PROJECTION_STYLES_STR {"parallel", "perspective"}

25  typedef enum { BackToFront, FrontToBack } CompositingStyles;
#define COMPOSITING_STYLES_STR {"BackToFront", "FrontToBack"}

typedef enum { NoShading, Phong, RefMap, RefVector } ShaderModes;
#define SHADER_MODES_STR {"NoShading", "Phong", "ReflectanceMap", "ReflectanceVector"}

30  typedef enum { Cube4Classic, Cube4Light } CubeModes;
#define CUBE_MODES_STR {"Cube4Classic", "Cube4Light"}

typedef enum { ViewFront, ViewLeft, ViewTop } MajorViewDirections;
#define MAJOR_VIEW_DIRECTION_STR {"ViewFront", "ViewLeft", "ViewTop"}

35  typedef enum { Left=0, Middle=1, Right=2 } XStepDirections;
#define X_STEP_DIRECTION_STR {"Left", "Middle", "Right"}
typedef enum { Up=0, /*Middle=1,*/ Down=2 } YStepDirections;
#define Y_STEP_DIRECTION_STR {"Up", "Middle", "Down"}

40  // Weight in Compos unit
static const int StepDirection[3] = {-1, 0, 1};

typedef enum { None=0, BackFace=1, TopBottomFace=2, SideFace=4 } RayEndFaces;
#define RAY_END_FACES_STR {"None", "BackFace", "TopBottomFace", "SideFace",
"SideFace"}

45  // VolVis SLC stuff
50

```

```

264
5   typedef enum { METER, MILLIMETER,           MICRON, FOOT, INCH } UnitType;
#define UNIT_TYPE_STR {"meter","millimeter","micron","foot","inch"}

10  typedef enum { BIORAD_CONFOCAL_DATA, MAGNETIC_RESONANCE_DATA,
COMPUTED_TOMOGRAPHY_DATA,
                                SIMULATION_DATA,
BINARY_VOXELIZED_DATA, FUZZY_VOXELIZED_DATA,
                                FUN_VOXELIZED_DATA, OTHER_DATA_ORIGIN
} DataOriginType;
#define DATA_ORIGIN_TYPE_STR {"Biorad confocal","MRI","CT","simulated", \
                           "binary voxelized","fuzzy voxelized","fun
voxelized","other origin"}

15  typedef enum { ORIGINAL_DATA, RESAMPLED_DATA, RESAMPLED_FILTERED_DATA,
                FILTERED_DATA,
OTHER_DATA_MODIFICATION } DataModificationType;
#define DATA_MODIFICATION_TYPE_STR {"original","resampled", \
                                   "resampled & filtered","filtered","other modified"}

20  typedef enum { NO_COMPRESSION, RUN_LENGTH_ENCODE } DataCompressionType;
#define DATA_COMPRESSION_TYPE_STR {"no compression","run length encoding"}

25  class TypeStr {
public:
// Cube4 stuff
    static const char * ProjectionStyle[];
    static const char * CompositingStyle[];
    static const char * ShaderMode[];
    static const char * CubeMode[];
    static const char * MajorViewDirection[];
    static const char * XStepDirection[];
    static const char * YStepDirection[];
    static const char * RayEndFace[];

30  // VolVis SLC stuff
    static const char * Unit[];
    static const char * DataOrigin[];
    static const char * DataModification[];
    static const char * DataCompression[];
};

35  // class TypeStr

40  #endif      // _Misc_h_
:::::::::::
cube4/ReflectanceMap.C
:::::::::::
// ReflectanceMap.C
// (c) Ingmar Bitter '97 / Urs Kanus '97

45  // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

#define WIN32

```

50

55

265

```

#include "RMap.h"
#include <strstrea.h>
5 #else
#include "ReflectanceMap.h"
#include <strstream.h>
#endif

10 #include <fstream.h>
#include <ctype.h>
#include <stdio.h> // sprintf

const double epsilon = .001;

15 void ReflectanceMap::Demo()
{
    Vector3D<double> view (1,0,0);
    Light source(1,0,0,1,1,1);
    DynaArray<Light> lightarray;
    lightarray.SetSize(1);
    20 lightarray[0] = source;

    ReflectanceMap rmap;
    cout << endl << "Demo of class " << typeid(rmap).name();
    cout << endl << "size : " << sizeof(ReflectanceMap) << " Bytes";
    cout << endl << "public member functions:";

25    cout << endl << "ReflectanceMap rmap;" ;
    cout << endl << "\t-----\n" << endl << rmap << "\t-----\n";
    cout << endl << "PerFrameSetup(";
    cout << endl << "\t\tconst int           newComponentAddressBits,";
    cout << endl << "\t\tconst bool          constAngResolution,";
    cout << endl << "\t\tVector3D<double> &view,";
30    cout << endl << "\t\tDynaArray<Light> &lightarray,";
    cout << endl << "\t\tconst double        specExp )";
    cout << endl << "PerFrameSetup(3,3,true,false,view,lightarr,2)";
    rmap.PerFrameSetup(3,3,true,false,view,lightarray,2);
    cout << endl << "\t-----\n" << endl << rmap << "\t-----\n";
    cout << endl << "virtual void LookUp(";
35    cout << endl << "\t\tconst Vector3D<FixPointNumber> gradientXYZ,";
    cout << endl << "\t\tconst bool         interpolateReflMap,";
    cout << endl << "\t\tconst bool         distortionCompensation,";
    cout << endl << "\t\tVector3D<FixPointNumber> & diffuseIntensity,";
    cout << endl << "\t\tVector3D<FixPointNumber> & specularIntensity )";
40    cout << endl << "virtual void WriteTable(const char * fileName)";
    cout << endl << "virtual void ReadTable (const char * fileName)";
    cout << endl << "virtual Vector3D<double> DiffuseIntensity( const
    Vector3D<double> & norm)";
    cout << endl << "virtual Vector3D<double> SpecularIntensity(const
    Vector3D<double> & norm)";
45    cout << endl << "void ShowValues( const char * fileName=\\"\\\" )";
    cout << endl;

    // ...
50

```

55

```

266
cout << endl << "End of demo of class " << typeid(rmap).name() << endl;
} // Demo
5
void ReflectanceMap::ShowMapImage(const bool refVectorLookup,
                                    const bool refMapInterpolation)
{
10
    cout << refVectorLookup << refMapInterpolation;
    int range = 0x1 << componentAddressBits;
    double du = 1.0/double(range - 1);
    double dv = 1.0/double(range - 1);
    double u, v;
    int colorPlaneOffset = 48 * range * range;
    int lineOffset = 8 * range;
    //int faceUSkip = 2 * range;
    int faceVSkip = 2 * range * lineOffset;
20
    Vector3D<FixPointNumber> diffuseIntensity; //, specularIntensity;
    char * mapImage = new char[48 * range * range * 3 - 1];
    int i,j;
    int faceUOffset = 0;
    int faceVOffset = 0;
25
    for ( i =-range+1,u=-1; i <= range; i++, u += du)
    { //cout << u << " ";
        for (j=-range+1, v = -1; j <= range; j++, v+=dv)
        { //cout << v << " ";
30
            Vector3D<FixPointNumber> NegNX( -1, u, v);
            DiffuseLookUp(NegNX, false, false, diffuseIntensity);
            mapImage[faceVSkip + faceVOffset*lineOffset +
                     faceUOffset - 1] =
                char(255*diffuseIntensity.R());
            mapImage[faceVSkip + faceVOffset*lineOffset + faceUOffset +
35
                     colorPlaneOffset - 1] =
                char(255*diffuseIntensity.G());
            mapImage[faceVSkip + faceVOffset*lineOffset + faceUOffset +
                     2*colorPlaneOffset - 1] =
                char(255*diffuseIntensity.B());
40
            /*
             Vector3D<FixPointNumber> NegNY( u,-1, -v);
             DiffuseLookUp(NegNY, false, false, diffuseIntensity);
             mapImage[faceVSkip + faceVOffset*lineOffset +
                     faceUSkip + faceUOffset] =
45
             char(255*diffuseIntensity.R());
             mapImage[faceVSkip + faceVOffset*lineOffset +
                     faceUSkip + faceUOffset +
                     colorPlaneOffset] =
             char(255*diffuseIntensity.G());
             mapImage[faceVSkip + faceVOffset*lineOffset +
50

```

```

267
    faceUSkip + faceUOffset +
2*colorPlaneOffset ] =
5   char(255*diffuseIntensity.B());

        Vector3D<FixPointNumber> PosNZ( u, -v, 1);
        DiffuseLookUp(PosNZ, false, false, diffuseIntensity);
        mapImage[2*faceVSkip + faceVOffset*lineOffset +
                  faceUSkip + faceUOffset] =
10  char(255*diffuseIntensity.R());
        mapImage[2*faceVSkip + faceVOffset*lineOffset +
                  faceUSkip + faceUOffset +
                  colorPlaneOffset] =
char(255*diffuseIntensity.G());
15   mapImage[2*faceVSkip + faceVOffset*lineOffset +
                  faceUSkip + faceUOffset +
                  2*colorPlaneOffset] =
char(255*diffuseIntensity.B());

20
        Vector3D<FixPointNumber> NegNZ( u, v, -1);
        DiffuseLookUp(NegNZ, false, false, diffuseIntensity);
        mapImage[faceVOffset*lineOffset +
                  faceUSkip + faceUOffset] =
char(255*diffuseIntensity.R());
25   mapImage[faceVOffset*lineOffset + faceUSkip +
                  faceUOffset + colorPlaneOffset] =
char(255*diffuseIntensity.G());
        mapImage[faceVOffset*lineOffset + faceUSkip +
                  faceUOffset + 2*colorPlaneOffset] =
char(255*diffuseIntensity.B());

30
        Vector3D<FixPointNumber> PosNX(1, -u, v);
        DiffuseLookUp(PosNX, false, false, diffuseIntensity);
        mapImage[faceVSkip + faceVOffset*lineOffset +
                  2*faceUSkip + faceUOffset] =
char(255*diffuseIntensity.R());
35   mapImage[faceVSkip + faceVOffset*lineOffset + 2*faceUSkip +
                  faceUOffset + colorPlaneOffset] =
char(255*diffuseIntensity.G());
        mapImage[faceVSkip + faceVOffset*lineOffset + 2*faceUSkip +
                  faceUOffset + 2*colorPlaneOffset] =
char(255*diffuseIntensity.B());

40

        Vector3D<FixPointNumber> PosNY( u, 1, v);
        DiffuseLookUp(PosNY, false, false, diffuseIntensity);
45   mapImage[faceVSkip + faceVOffset*lineOffset +
                  3*faceUSkip + faceUOffset] =
char(255*diffuseIntensity.R());
        mapImage[faceVSkip + faceVOffset*lineOffset + 3*faceUSkip +
                  faceUOffset + colorPlaneOffset] =
char(255*diffuseIntensity.G());
50   mapImage[faceVSkip + faceVOffset*lineOffset + 3*faceUSkip +

```

```

268
= char(255*diffuseIntensity.B());
      faceUOffset + 2*colorPlaneOffset]
5           */
           faceVOffset++;
}
faceVOffset = 0;
faceUOffset++;
}
10
// now write the image to a file

char mapFileName[50];
sprintf(mapFileName, "img/refMapImage.rgb");
ofstream mapImageFile(mapFileName);
15
mapImageFile.write(mapImage, 3 * colorPlaneOffset);

char str[200];
sprintf(str, "convert -size %ix%i %s %s \n",
       8*range, 6*range,
20       "img/refMapImage.rgb",
       "img/refMapImage.miff");
system(str);

)
25

void ReflectanceMap::ShowValues(const char * fileName, int color)
{
    ofstream output(fileName);
    int range = 0x1 << componentAddressBits;

    double du = 1.0/double(range - 1);
    double dv = 1.0/double(range - 1);
    double u, v;
    Vector3D<FixPointNumber> diffuseIntensity; //, specularIntensity;
35
    strstream posXfaceR; //, posXfaceG, posXfaceB,
    strstream posYfaceR; //, posYfaceG, posYfaceB,
    strstream posZfaceR; //, posZfaceG, posZfaceB,
    strstream negXfaceR; //, negXfaceG, negXfaceB,
    strstream negYfaceR; //, negYfaceG, negYfaceB,
40
    strstream negZfaceR; //, negZfaceG, negZfaceB;
    int i,j;
    int prec = 4;
    int wid = prec + 2;
    Vector3D<FixPointNumber> filter;
    switch (color) {
45
    case 0:
        filter(1,0,0); break;
    case 1:
        filter(0,1,0); break;
50
}

```

55

```

269
case 2:
    filter(0,0,1); break;
5    default:
        return;
    }

10   char *filler = new char[(2*range)*(wid+1) +1];
for ( i=0; i<(2*range)*(wid+1) +1; i++) filler[i] = ' ';
filler[2*range*(wid+1)] = '\0';

15   for ( i =-range+1, u=-1; i <= range; i++, u += du)
{ //cout << u << " ";
    for (j=-range+1, v = -1; j <= range; j++, v+=dv)
{ //cout << v << " ";

20     Vector3D<FixPointNumber> NegNX( -1, u, v);
     DiffuseLookUp(NegNX, false, false, diffuseIntensity);

25     posYfaceR.precision(prec); posYfaceR.width(wid);
     posYfaceR << (double)(diffuseIntensity*filter) << " ";

30     Vector3D<FixPointNumber> NegNY( u,-1, -v);
     DiffuseLookUp(NegNY, false, false, diffuseIntensity);
     negYfaceR.precision(prec); negYfaceR.width(wid);
     negYfaceR << (double)(diffuseIntensity*filter) << " ";

35     Vector3D<FixPointNumber> PosNZ( u, -v, 1);
     DiffuseLookUp(PosNZ, false, false, diffuseIntensity);
     posZfaceR.precision(prec); posZfaceR.width(wid);
     posZfaceR << (double)(diffuseIntensity*filter) << " ";

40     Vector3D<FixPointNumber> NegNZ( u, v, -1);
     DiffuseLookUp(NegNZ, false, false, diffuseIntensity);
     negZfaceR.precision(prec); negZfaceR.width(wid);
     negZfaceR << (double)(diffuseIntensity*filter) << " ";

45     }
    for (j=-range+1, v = -1; j <= range; j++, v+=dv)
    {
        Vector3D<FixPointNumber> PosNY( u, 1, v);
        DiffuseLookUp(PosNY, false, false, diffuseIntensity);
        posYfaceR.precision(prec); posYfaceR.width(wid);
        posYfaceR << (double)(diffuseIntensity*filter) << " ";
    }

50     for (j=-range+1, v = -1; j <= range; j++, v+=dv)
    {

```

```

5           Vector3D<FixPointNumber> PosNX(1, -u, v);
DiffuseLookUp(PosNX, false, false, diffuseIntensity);
posYfaceR.precision(prec); posYfaceR.width(wid);
posYfaceR << (double)(diffuseIntensity*filter) << " ";
10
}
posXfaceR << "\n";
posYfaceR << "\n";
posZfaceR << "\n";
negXfaceR << "\n";
negYfaceR << "\n";
negZfaceR << "\n";
15
}
posXfaceR << ends;
posYfaceR << ends;
posZfaceR << ends;
negXfaceR << ends;
negYfaceR << ends;
negZfaceR << ends;
20
/*
25   cout << negZfaceR.str();
   cout << filler << "-----\n";
   cout << posYfaceR.str();
   cout << filler << "-----\n";
   cout << posZfaceR.str();
   cout << filler << "-----\n";
30   cout << negYfaceR.str();
*/
   output << negZfaceR.str();
   output << filler << "-----\n";
   output << posYfaceR.str();
   output << filler << "-----\n";
35   output << posZfaceR.str();
   output << filler << "-----\n";
   output << negYfaceR.str();

   output.close();
40   delete [] filler;
}

///////////////////////////////
// Construction/Destruction
45 ///////////////////////

```

```

ReflectanceMap::ReflectanceMap()
{
    componentAddressBits = 0;
50   pDiffuseIntensities = 0;
   pSpecularIntensities = 0;
   //Light source(1.0,0.0,0.0);
}

```

```

271
    lightSource.SetSize(0);
    //lightSource[0] = source;
5   }

ReflectanceMap::~ReflectanceMap()
{
10  }

void ReflectanceMap::PerFrameSetup(const int newComponentAddressBits,
15
                           const int newMapWeightBits,
                           const bool constantAngularResolution,
                           const bool refVectorLookup,
20
                           const Vector3D<double> & newViewVector,
                           const DynaArray<Light> & newLightSource,
                           const double newSpecularExponent)
25
{
    enum {NO_CHANGE, DIFFUSE, SPECULAR};

30
    int state1 = NO_CHANGE;
    int state2 = NO_CHANGE;

    static int range;
    mapWeightBits = newMapWeightBits;

35
    if (componentAddressBits != newComponentAddressBits) {
        state1 = DIFFUSE;
        state2 = SPECULAR;
        componentAddressBits = newComponentAddressBits;
    }

40
    range = 0x1 << componentAddressBits;

    delete [] pDiffuseIntensities;
    delete [] pSpecularIntensities;

45
    // map has one section for each maximum component x,y, or z
    // each section has 8 subsections (4 quadrants, 2 faces)
    // each subsection contains range * range values
    pDiffuseIntensities = new rgb[range * range * 8 * 3];
    pSpecularIntensities = new rgb[range * range * 8 * 3];
}
50

// Handle changing light sources here

```

```

272
if(lightSource.Size() != newLightSource.Size())
{
    state1 = DIFFUSE;
    state2 = SPECULAR;
}
else
{
    for (int i = 0; i < lightSource.Size(); i++)
    {
        if(lightSource[i].Direction() != newLightSource[i].Direction() ||
           lightSource[i].Intensity() != newLightSource[i].Intensity())
        {
            state1 = DIFFUSE;
            state2 = SPECULAR;
            break;
        }
    }
    if( state1 == NO_CHANGE) {
        Vector3D<double> temp = viewVector - newViewVector;

        // if we use reflection vector lookup, view vector change
        // doesn't matter
        if (!refVectorLookup){
            state2 = (temp.Norm() > epsilon ) ? SPECULAR : state2;
        }
        state2 = (specularExponent - newSpecularExponent > epsilon)
                 ? SPECULAR : state2;
    }

    if(state1 == NO_CHANGE && state2 == NO_CHANGE) return;

    viewVector = newViewVector;
    specularExponent = newSpecularExponent;
    lightSource = newLightSource;
    int mapOffset = range*range*8;
    short sign1, sign2;
    int signAddress;

    // alpha between 0 to 45 degrees
    double alphaIncrement = (PI / 4.0) / double(range - 1);
    double alphaU = 0.0;
    double alphaV = 0.0;

    // or range-1 steps between 0 and 1
    double dU = 1.0/double(range - 1);
    double dV = 1.0/double(range - 1);

    double u = 0.0;
    double v = 0.0;
}

```

55

```

for ( int quadrant = 0; quadrant < 4; quadrant++){
    for ( int axis1 = 0; axis1 < range; axis1++){
        for ( int axis2 = 0; axis2 < range; axis2++){

            // we can either have constant stepping along the cube
5           face
            // or constant angular stepping of the normals
10          if (constantAngularResolution){
                u = tan(alphaU);
                v = tan(alphaV);
            }

15          // we go ^+
//           |
//           1   |   0
//           -----+----> +
//           2   |   3
20          //
switch (quadrant){
case 0: sign1 = 1; sign2 = 1; break;
case 1: sign1 = -1; sign2 = 1; break;
case 2: sign1 = -1; sign2 = -1; break;
case 3: sign1 = 1; sign2 = -1; break;
25

30          Vector3D<double> PosNX(      1, sign1*u, sign2*v);
          Vector3D<double> NegNX(     -1, sign1*u, sign2*v);
          Vector3D<double> PosNY( sign1*u,      1, sign2*v);
          Vector3D<double> NegNY( sign1*u,     -1, sign2*v);
          Vector3D<double> PosNZ( sign1*u, sign2*v,      1);
          Vector3D<double> NegNZ( sign1*u, sign2*v,     -1);

35          Vector3D<double> temp;
          char signs;

40          if(state1 == DIFFUSE){

              temp = DiffuseIntensity(PosNX);
              signs =
                  2 * (sign1<0) +
                  4 * (sign2<0);
              signAddress = signs<<(2*componentAddressBits);

45          pDiffuseIntensities[signAddress+axis2*range+axis1](temp.R(),
              temp.G(),
              temp.B());
50

```

```

274
temp = DiffuseIntensity(PosNY);
signs =
5      1 * (sign1<0) +
        4 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);

pDiffuseIntensities[mapOffset+signAddress+axis2*range+axis1](temp.R(),
10
                           temp.G(),
                           temp.B());

15
temp = DiffuseIntensity(PosNZ);
signs =
20      1 * (sign1<0) +
        2 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);

pDiffuseIntensities[2*mapOffset+signAddress+axis2*range+axis1](temp.R(),
25
                           temp.G(),
                           temp.B());

30
temp = DiffuseIntensity(NegNX);
signs = 1 +
        2 * (sign1<0) +
        4 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);

35
pDiffuseIntensities[signAddress+axis2*range+axis1](temp.R(),
                           temp.G(),
                           temp.B());

40
                           temp.B());

temp = DiffuseIntensity(NegNY);
45
signs = 2 +
        1 * (sign1<0) +
        4 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);

50
pDiffuseIntensities[mapOffset+signAddress+axis2*range+axis1](temp.R(),
                           temp.G(),
                           temp.B());

```

55

```

5                                     temp.B()));

temp = DiffuseIntensity(NegNZ);
signs = 4 +
       1 * (sign1<0) +
       2 * (sign2<0);
10 signAddress = signs<<(2*componentAddressBits);

pDiffuseIntensities[2*mapOffset+signAddress+axis2*range+axis1](temp.R(),

15                                     temp.G(),

                                     temp.B()));

20 }
if( state2 == SPECULAR ) {

25 temp = SpecularIntensity(PosNX, refVectorLookup);
signs =
       2 * (sign1<0) +
       4 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);
30 pSpecularIntensities[signAddress+axis2*range+axis1](temp.R(),

                                     temp.G(),
35                                     temp.B()));

temp = SpecularIntensity(PosNY, refVectorLookup);
signs =
       1 * (sign1<0) +
       4 * (sign2<0);
signAddress = signs<<(2*componentAddressBits);
40 pSpecularIntensities[mapOffset+signAddress+axis2*range+axis1](temp.R(),
45                                     temp.G(),

                                     temp.B()));

50 temp = SpecularIntensity(PosNZ, refVectorLookup);
signs =

```

```

      276
      1 * (sign1<0) +
      2 * (sign2<0);
5     signAddress = signs<<(2*componentAddressBits);

pSpecularIntensities[2*mapOffset+signAddress+axis2*range+axis1](temp.R(),
                                                              
10    temp.G(),
                                                              
                                                              
15    temp = SpecularIntensity(NegNX, refVectorLookup);
    signs = 1 +
            2 * (sign1<0) +
            4 * (sign2<0);
    signAddress = signs<<(2*componentAddressBits);

20   pSpecularIntensities[signAddress+axis2*range+axis1](temp.R(),
                                                              
                                                              
25   temp.G(),
                                                              
                                                              
25   temp.B();
    temp = SpecularIntensity(NegNY, refVectorLookup);
    signs = 2 +
            1 * (sign1<0) +
            4 * (sign2<0);
30   signAddress = signs<<(2*componentAddressBits);

pSpecularIntensities[mapOffset+signAddress+axis2*range+axis1](temp.R(),
                                                              
35   temp.G(),
                                                              
                                                              
40   temp.B();
    temp = SpecularIntensity(NegNZ, refVectorLookup);
    signs = 4 +
            1 * (sign1<0) +
            2 * (sign2<0);
    signAddress = signs<<(2*componentAddressBits);

45   pSpecularIntensities[2*mapOffset+signAddress+axis2*range+axis1](temp.R(),
                                                              
                                                              
50   temp.G(),
                                                              
                                                              
55   temp.B());

```

```

277
)
if (axis2 == range-1){
    v = 0.0;
    alphaV = 0.0;
}
else{
    v += dV;
    alphaV += alphaIncrement;
}
)
if (axis1 == range-1){
    u = 0.0;
    alphaU = 0.0;
}
else{
    u += dU;
    alphaU += alphaIncrement;
}
)
}

25 void ReflectanceMap::WriteTable(const char * fileName)
{
    ofstream output(fileName);
    output << "Reflectance Map File\n";
    output << *this;
    output << "\n\n";
    int range = 0x1 << componentAddressBits;

    int arraySize = range * range * 8 * 3; // See explanation in
    PerFrameSetup
    output << "\ndiffuse intensities\n\n";

    for( int i=0; i < arraySize; i++)
    {
        output << pDiffuseIntensities[i].R() << " " ;
        output << pDiffuseIntensities[i].G() << " " ;
        output << pDiffuseIntensities[i].B() << " ";
    }
    output << "\nspecular intensities\n\n";

    for( int i=0; i < arraySize; i++)
        output << pSpecularIntensities[i].R() << " " <<
            pSpecularIntensities[i].G() << " " <<
            pSpecularIntensities[i].B() << " ";
    output.close();
}

```

55

```

    }

5   void skipAlpha( ifstream input)
{
    cout << input;
    /*    int c;
10   while ((c = input.get()) != EOF)
    {
        if (!isdigit(c) && c != '-')
            continue;

        input.putback(c);
15   break;
    }
}

20 void ReflectanceMap::ReadTable (const char * fileName)
{
    cout << fileName;
    /*    ifstream input(fileName);

25   skipAlpha(input);
    input >> componentAddressBits;
    skipAlpha(input);

30   double x,y,z,r,g,b;
    input>> x; input.get();
    input>> y; input.get();
    input>> z;
    viewVector(x,y,z);

35   skipAlpha(input);
    int numLights;
    input >> numLights;
    skipAlpha(input);

40   lightSource.SetSize(numLights);
    Light source;
    for (int i=0; i< numLights; i++)
    {
45     input >> x; input.get();
        input >> y; input.get();
        input >> z; skipAlpha(input);
        input >> r; input.get();
        input >> g; input.get();
        input >> b; skipAlpha(input);
50     source(x,y,z,r,g,b,1);
        lightSource[i] = source;
    }
}

```

55

```

    input >> specularExponent;
5
    skipAlpha(input);

    delete [] pDiffuseIntensities;
    delete [] pSpecularIntensities;
10
    int range = 0x1 << componentAddressBits;
    int arraySize = range * range * 8 * 3; // See explanation in
    PerFrameSetup
    pDiffuseIntensities = new rgb[range * range * 8 * 3];
    pSpecularIntensities = new rgb[range * range * 8 * 3];
15
    for (i=0; i<arraySize; i++)
    {
        input >> r;
        input >> g;
        input >> b;
        pDiffuseIntensities[i](r,g,b);
20
    }
    skipAlpha(input);

    for (i=0; i<arraySize; i++)
    {
30
        input >> r;
        input >> g;
        input >> b;
        pSpecularIntensities[i](r,g,b);
    }
35
    input.close();
    */
40
}
45
// show/set data & data properties
//
50
ostream & ReflectanceMap::Ostream(ostream & os) const
{
    // append Light info to os
    os << "Bits in Gradient\t\t" << componentAddressBits << "\n";
    os << "View Vector\t\t\t" << viewVector << "\n";
    os << "Number of Light Sources\t\t" << lightSource.Size() << "\n";
    for (int i=0; i < lightSource.Size(); i++)
        os << "Light source " << "\t\t\t" << lightSource[i] << "\n";
}
55

```

```

5
// return complete os
return os;

10
} // Ostream

15
Vector3D<double> ReflectanceMap::DiffuseIntensity(const Vector3D<double> & norm)
{
    Vector3D<double> intensitySum(0,0,0), lightVector, surfaceNormal;
    double n;

    n = norm.Norm();
    if (n>0) {
        surfaceNormal = norm/n;
    }

20
    for (int i=0; i < lightSource.Size(); i++) {

        lightVector=lightSource[i].Direction();
        lightVector /= lightVector.Norm();

        double NL = surfaceNormal * lightVector;
        if (NL < 0)
            NL = 0;
        else
            NL = pow(NL, lightSource[i].Sharpness());

        intensitySum += (lightSource[i].Intensity()) * NL;
    }
    return intensitySum;

30
} // DiffuseIntensity

35
Vector3D<double> ReflectanceMap::SpecularIntensity(const Vector3D<double> &
vector,
40
        const
        bool refVectorLookup)
{
    Vector3D<double> intensitySum(0,0,0), lightVector;
    Vector3D<double> surfaceNormal, lightReflection;
    Vector3D<double> eyeReflection;
    double n, RL, RV, NL, phongComponent;

45
    if (refVectorLookup){ // vector is reflectionVector
        n = vector.Norm();
        if (n>0){
            RL = dot(surfaceNormal, vector);
            RV = dot(surfaceNormal, eyeReflection);
            NL = dot(surfaceNormal, lightVector);
            phongComponent = RL * RV / (NL * RL);
        }
    }
}

```

```

281
    eyeReflection =      vector / n;
}
5
for (int light=0; light<lightSource.Size(); ++light) {

    lightVector = lightSource[light].Direction();
    lightVector /= lightVector.Norm();
10
    RL = eyeReflection * lightVector;

    if (RL < 0)
        phongComponent = 0;
15
    else
        phongComponent = pow(RL, specularExponent);

    intensitySum +=
        lightSource[light].Intensity() * phongComponent;
20
}
20
else{ // vector is surface normal

    n = vector.Norm();
25
    if (n>0) {
        surfaceNormal = vector/n;
    }

    for (int light=0; light<lightSource.Size(); ++light) {

        lightVector = lightSource[light].Direction();
        lightVector /= lightVector.Norm();

        NL = surfaceNormal * lightVector;

        lightReflection = surfaceNormal;
        lightReflection *= (2*NL);
        lightReflection = lightReflection - lightVector;

        RV = lightReflection * viewVector;
40
        if (RV < 0)
            phongComponent = 0;
        else
            phongComponent = pow(RV, specularExponent);

        intensitySum +=    lightSource[light].Intensity() *
45
        phongComponent;
    }
}

50
    return intensitySum;
} // SpecularIntensity

```

282

```

void
ReflectanceMap::ComputeLookupVector(const Vector3D<FixPointNumber> vectorXYZ,
5
                                         FixPointNumber & index1,
                                         FixPointNumber & index2,
                                         int & mapAddress)
10  {

    FixPointNumber maxComponent, vector1, vector2;
    Vector3D<FixPointNumber> vector;

15  int range = 1<<componentAddressBits;
    int faceQuadrant = range * range;
    int frontAndBackFace = faceQuadrant * 8;
    int mapOffset;

    short gradientInputPrecision = vectorXYZ.X().FractionBits();

20  // extract sign bits
    int signs =
        1 * (vectorXYZ.X()<0) +
        2 * (vectorXYZ.Y()<0) +
        4 * (vectorXYZ.Z()<0);

25  vector((FixPointNumber)(vectorXYZ.X().Abs()),
             (FixPointNumber)(vectorXYZ.Y().Abs()),
             (FixPointNumber)(vectorXYZ.Z().Abs()));

    // Sort
30  bool yGreaterX, zGreaterX, zGreaterY;
    bool equal;

    yGreaterX = (vector.X() < vector.Y());
    zGreaterX = (vector.X() < vector.Z());
35  zGreaterY = (vector.Y() < vector.Z());

    equal = (!yGreaterX && !zGreaterX && !zGreaterY);

    if (zGreaterX && zGreaterY) {
40      maxComponent = vector.Z();
      vector1 = vector.X();
      vector2 = vector.Y();
      mapOffset = 2 * frontAndBackFace; // address upper third of table
    }
    else if (zGreaterX && !zGreaterY) {
45      if (yGreaterX) {
          maxComponent = vector.Y();
          vector1 = vector.X();
          vector2 = vector.Z();
          mapOffset = frontAndBackFace; // address middle third of table
        }
50

```

55

283

```

}
else{
    maxComponent = vector.Z();
    vector1 = vector.X();
    vector2 = vector.Y();
    mapOffset = 2 * frontAndBackFace; // address upper third of
table
}
10 else if (!zGreaterX && zGreaterY) {
    if (yGreaterX) {
        maxComponent = vector.Z();
        vector1 = vector.X();
        vector2 = vector.Y();
    15     mapOffset = 2 * frontAndBackFace; // address upper third of
table
    }
    else {
        maxComponent = vector.X();
        vector1 = vector.Y();
        vector2 = vector.Z();
        mapOffset = 0; // lower third of table
    }
}
20 else if (!zGreaterX && !zGreaterY) {
    if (yGreaterX) {
        maxComponent = vector.Y();
        vector1 = vector.X();
        vector2 = vector.Z();
        mapOffset = frontAndBackFace; // address middle third of table
    }
}
25 else {
    maxComponent = vector.X();
    vector1 = vector.Y();
    vector2 = vector.Z();
    mapOffset = 0; // lower third of table
}
30
}
35 else if (equal) {
    maxComponent = vector.X();
    vector1 = vector.Y();
    vector2 = vector.Z();
    mapOffset = 0; // lower third of table
    cout << endl << "equal :" << maxComponent << " " << vector1 << " " << vector2;
}
40
}

// Shift
if (maxComponent>0){
    int compare = 1<<(gradientInputPrecision - 1);
    int bitNumber = gradientInputPrecision - 1;
    bool done = false;

    // all significant bits after decimal point
}
45

```

50

55

```

284
maxComponent = maxComponent>>gradientInputPrecision;
vector1 = vector1>>gradientInputPrecision;
vector2 = vector2>>gradientInputPrecision;

5      while (bitNumber>0 && !done) {
        if (!(maxComponent & compare) &&
            !(vector1 & compare) &&
            !(vector2 & compare))){
10         maxComponent = maxComponent<<1;
            vector1 = vector1<<1;
            vector2 = vector2<<1;
        }
        else
15         done = true;
        bitNumber--;
    }

20 // Normalization
if (maxComponent>0){

    vector1 /= maxComponent;
    vector2 /= maxComponent;
}

25
index1 = vector1;
index2 = vector2;

signs = signs * faceQuadrant;
30 mapAddress = mapOffset + signs;
}

void ReflectanceMap::DiffuseLookUp(const Vector3D<FixPointNumber> gradientXYZ,
35
        const bool interpolateReflMap,
        const bool distortionCompensation,
        Vector3D<FixPointNumber> & diffuseIntensity)

40 {
    FixPointNumber index1;
    FixPointNumber index2;
    int mapAddress;
    int range = 1<<componentAddressBits;
    double alphaIncrementReciprocal = double(range - 1)/(PI/4.0);
45
    ComputeLookupVector(gradientXYZ, index1, index2, mapAddress);

    if (!distortionCompensation){
        // We don't want 1.0 if we don't use distortion compensation
        index1 *= (double(range-1)/double(range));
50
}

```

```

      285
    index2 *= (double(range-      1)/double(range));
}
5
// Distortion compensation computed on the fly
if (distortionCompensation){
    index1 = atan(index1) * alphaIncrementReciprocal;
    index2 = atan(index2) * alphaIncrementReciprocal;
}
10
else{
    index1 = index1<<componentAddressBits;
    index2 = index2<<componentAddressBits;
}

15
// Reflectance map address and interpolation weights
char addr1 = int(index1);
char addr2 = int(index2);

FixPointNumber weight1 = index1 - FixPointNumber(addr1);
20
weight1 = weight1.Round(mapWeightBits);

FixPointNumber weight2 = index2 - FixPointNumber(addr2);
weight2 = weight2.Round(mapWeightBits);

25
// Map Access & Interpolation (diffuse component)
if (interpolateReflMap) {
    Vector3D<FixPointNumber> intensityBase, intensityRight;
    Vector3D<FixPointNumber> intensityTopLeft, intensityTopRight;
    Vector3D<FixPointNumber> top, bottom;
}

30
//interpolate diffuse part
intensityBase = pDiffuseIntensities[mapAddress+addr2*range+addr1];
intensityRight =
pDiffuseIntensities[mapAddress+addr2*range+addr1+1];
intensityTopLeft =
35
pDiffuseIntensities[mapAddress+(addr2+1)*range+addr1];
intensityTopRight =
pDiffuseIntensities[mapAddress+(addr2+1)*range+addr1+1];

bottom = intensityRight - intensityBase;
bottom *= weight1;
40
bottom += intensityBase;

top = intensityTopRight - intensityTopLeft;
top *= weight1;
top += intensityTopLeft;

45
diffuseIntensity = top - bottom;
diffuseIntensity *= weight2;
diffuseIntensity += bottom;

50
}
else{

```

55

286

```

    diffuseIntensity =
pDiffuseIntensities[mapAddress+addr2*range+addr1];
5
}
}

void ReflectanceMap::SpecularLookUp(const Vector3D<FixPointNumber> vectorXYZ,
10
                           const bool interpolateReflMap,
                           const bool distortionCompensation,
                           Vector3D<FixPointNumber> & specularIntensity)
15
{
    FixPointNumber index1;
    FixPointNumber index2;
    int mapAddress;
    int range = 1<<componentAddressBits;
20
    double alphaIncrementReciprocal = double(range - 1)/(PI/4.0);

    ComputeLookupVector(vectorXYZ, index1, index2, mapAddress);

    if (!distortionCompensation){
        // We don't want 1.0 if we don't use distortion compensation
        index1 *= (double(range-1)/double(range));
        index2 *= (double(range-1)/double(range));
    }

30
    // Distortion compensation computed on the fly
    if (distortionCompensation){
        index1 = atan(index1) * alphaIncrementReciprocal;
        index2 = atan(index2) * alphaIncrementReciprocal;
    }
35
    else{
        index1 = index1<<componentAddressBits;
        index2 = index2<<componentAddressBits;
    }

    // Reflectance map address and interpolation weights
40
    char addr1 = int(index1);
    char addr2 = int(index2);

    FixPointNumber weight1 = index1 - FixPointNumber(addr1);
    weight1 = weight1.Round(mapWeightBits);

45
    FixPointNumber weight2 = index2 - FixPointNumber(addr2);
    weight2 = weight2.Round(mapWeightBits);

    // Map Access & Interpolation (specular component)
50
    if (interpolateReflMap) {

```

55

```

287
Vector3D<FixPointNumber> intensityBase, intensityRight;
Vector3D<FixPointNumber> intensityTopLeft, intensityTopRight;
Vector3D<FixPointNumber> top, bottom;

5 // only for debugging: report address overflow
if ((addr1 > (range - 2)) || (addr2 > (range - 2))){
    cout << endl << "Address-Overflow !!!(addr1 = "<<addr1<<, addr2
= ";
10 cout << addr2;
}

// interpolate specular part
intensityBase = pSpecularIntensities[mapAddress+addr2*range+addr1];
intensityRight =
15 pSpecularIntensities[mapAddress+addr2*range+addr1+1];
intensityTopLeft =
pSpecularIntensities[mapAddress+(addr2+1)*range+addr1];
intensityTopRight =
pSpecularIntensities[mapAddress+(addr2+1)*range+addr1+1];

20 bottom = intensityRight - intensityBase;
bottom *= weight1;
bottom += intensityBase;

top = intensityTopRight - intensityTopLeft;
25 top *= weight1;
top += intensityTopLeft;

specularIntensity = top - bottom;
specularIntensity *= weight2;
specularIntensity += bottom;
30 }

35 else{
    specularIntensity =
pSpecularIntensities[mapAddress+addr2*range+addr1];
}

40 ::::::::::::::
cube4/ReflectanceMap.h
:::::::::::
// ReflectanceMap.h
// Ingmar Bitter '97 / Urs Kanus '97

45 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "Misc.h"
50

```

55

288

```

#include "Object.h"
#include "FixPointNumber.h"
#include "Vector3D.h"
#include "DynaArray.h"
#include "Light.h"

#define PI 3.1415927

10 class ReflectanceMap : virtual public Object {
public:

    static void      Demo ();

15    // constructors & destructors
    ReflectanceMap ();
    ~ReflectanceMap ();

    // show/set data & data properties
    // - class Object requirements
20    virtual ostream & Ostream (ostream & ) const;

    // - local show/set functions
    virtual void PerFrameSetup(const int newComponentAddressBits,

25    const int newMapWeightBits,

        const bool constantAngularResolution,
        const bool refVectorLookup,
30    const Vector3D<double> & newViewVector,
        const DynaArray<Light> & newLightSource,
        const double newSpecularExponent);

35    virtual void ComputeLookupVector(const Vector3D<FixPointNumber> vectorXYZ,
        FixPointNumber & index1,
        FixPointNumber & index2,
40        int & mapAddress);

        virtual void DiffuseLookUp(const Vector3D<FixPointNumber> gradientXYZ,
        const bool interpolateReflMap,
45        const bool distortionCompensation,
        Vector3D<FixPointNumber> & diffuseIntensity);

50
55

```

```

virtual void SpecularLookUp(const Vector3D<FixPointNumber> vectorXYZ,
5           const bool interpolateReflMap,
           const bool distortionCompensation,
           Vector3D<FixPointNumber> & specularIntensity);
10

virtual void WriteTable(const char * fileName);
virtual void ReadTable (const char * fileName);
virtual Vector3D<double> DiffuseIntensity( const Vector3D<double> & norm);
virtual Vector3D<double> SpecularIntensity(const Vector3D<double> & norm,
15
                           const bool
refVectorLookup);

void ShowValues( const char * fileName, int color ); // 0-red 1-green
20 2-blue
void ShowMapImage( const bool refVectorLookup,
                   const bool
refMapInterpolation);

protected:
25     int          componentAddressBits;
     int          mapWeightBits;
     Vector3D<double> viewVector;
     Vector3D<double> IAmbient;
     DynaArray<Light> lightSource;
     double        specularExponent;
30

typedef     Vector3D<FixPointNumber> rgb;
rgb*          pDiffuseIntensities;
rgb*          pSpecularIntensities;
};

35 #endif      // _ReflectanceMap_h_
:::::::::::
cube4/Shadel.C
:::::::::::
// Shadel.C
40 // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

#include "Shadel.h"
45

Shadel::Shadel(FixPointNumber setR, FixPointNumber setG, FixPointNumber setB,
               FixPointNumber setA)
: r(setR), g(setG), b(setB), a(setA)
{
50

```

```

290
} // constructor

5
Shadel::Shadel(const Shadel & shadel)
    : r(shadel.r),g(shadel.g),b(shadel.b),a(shadel.a)
{
} // constructor

10
// friend output function
ostream & operator << (ostream & os, const Shadel & s)
{ return os<<"("<<s.r<<","<<s.g<<","<<s.b<<","<<s.a<<")"; }
::::::::::
15
cube4/Shadel.h
::::::::::
// Shadel.h
// (c) Ingmar Bitter '97

20
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _Shadel_h_      // prevent multiple includes
#define _Shadel_h_

25
#include <iostream.h> // cout
#include "FixPointNumber.h"

class Shadel {
public: // data
30
    FixPointNumber r,g,b,a;
public: // member functions
    Shadel(FixPointNumber r=0, FixPointNumber g=0, FixPointNumber b=0,
            FixPointNumber a=0);
    Shadel(const Shadel & shadel);
35
public: // friend functions
    friend ostream & operator << (ostream & os, const Shadel & shadel);
}; // Shadel

#endif // _Shadel_h_
40
::::::::::
cube4/ShaderPipeline.C
::::::::::
// ShaderPipeline.C
// (c) Ingmar Bitter '97 / Urs Kanus '97

45
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "ShaderPipeline.h"

50
void ShaderPipeline::Demo()
{
    ShaderPipeline shader;

```

55

```

291
cout << endl << "Demo of class " << typeid(shader).name();
cout << endl << "size : " << sizeof(ShaderPipeline) << " Bytes";
cout << endl << "public member functions:";
cout << endl << "ShaderPipeline shader; = " << shader;
cout << endl << "End of demo of class " << typeid(shader).name() << endl;
} // Demo

10   /////////////////////////////////
// constructors & destructors

// static first init
int    ShaderPipeline::numOfChips          = 0;
int    ShaderPipeline::numOfPipelinesPerChip = 0;
15   Cube4 *ShaderPipeline::cube4           = 0;

ShaderPipeline::ShaderPipeline()
{
} // constructor

20
ShaderPipeline::~ShaderPipeline()
{
} // destructor

25
///////////////////////////////
// show/set data & data properties

30   ostream & ShaderPipeline::Ostream(ostream & os) const
{
    // append ShaderPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
35   os << endl << "  chipIndex           = " << chipIndex;

    // return complete os
    return os;
}

40
} // Ostream

45
///////////////////////////////
// show/set data & data properties
//
// - local show/set functions

void ShaderPipeline::GlobalSetup(const int setNumOfChips,
50
        const int setNumOfPipelinesPerChip)

```

292

```

5
    {
        numOfChips          = setNumOfChips;
        numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    } // GlobalSetup

10
void ShaderPipeline::LocalSetup(const int setChipIndex,
                               const int setPipelineIndex,
                               ShaderStage & shaderStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.voxel = &(shaderStage.inputs.voxel[pipelineIndex]);
    inputs.gx = &(shaderStage.inputs.gx[pipelineIndex]);
    inputs gy = &(shaderStage.inputs.gy[pipelineIndex]);
    inputs.gz = &(shaderStage.inputs.gz[pipelineIndex]);
20
    inputs.weightsXYZ = &(shaderStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = shaderStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(shaderStage.inputs.perPipelineControlFlags[pipelineIndex]);

25
    results.shadel = &(shaderStage.results.shadel[pipelineIndex]);
    results.weightsXYZ = &(shaderStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        = &(shaderStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(shaderStage.results.perChipControlFlags);

30
    cube4 = shaderStage(cube4);

    reflectanceMap = &(shaderStage.reflectanceMap);
    colorLUT = &(shaderStage.colorLUT);
} // LocalSetup

35
void ShaderPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in ShaderStage
} // PerFrameSetup

40
///////////////////////////////
45
// local computation functions
void ShaderPipeline::RunForOneClockCycle()
{
/*
    // get inputs and transform to 0 ... 1 range
    int index      = inputs.voxel->raw16bit;
50

```

55

```

293
double gx      = *inputs.gx / 255.0;
double gy      = *inputs.gy / 255.0;
double gz      = *inputs.gz / 255.0;
5

// color transferfunction
double red = double((colorLUT->Red(index))/255.0);
double green = double((colorLUT->Green(index))/255.0);
double blue = double((colorLUT->Blue(index))/255.0);

10
//double gradientMagnitude = sqrt(gx*gx + gy*gy + gz*gz);
//double noiseLevel(0.05);
//double signalToNoiseGradientMagnitude =
//Min(1.0, gradientMagnitude / noiseLevel);

15
Vector3D<double> viewVector;
Vector3D<double> diffuseIntensity;
Vector3D<double> specularIntensity;

20
if (cube4->shaderMode == Phong) {

    // Phong shading, assumes lightsource and eyepoint at infinite
distance

    //Vector3D<double> halfway;
25
    Vector3D<double> lightReflection;
    Vector3D<double> surfaceNormal;
        Vector3D<double> lightVector;
        //double norm;

        surfaceNormal(-gx, -gy, -gz);
30
        surfaceNormal /= surfaceNormal.Norm();

        // viewVector is inverse of sightRay
        viewVector( (double) cube4->viewPointXYZ.X() - (double) cube4-
>datasetSizeXYZ.X() / 2.0,
            (double) cube4->viewPointXYZ.Y()
35
            - (double) cube4->datasetSizeXYZ.Y() / 2.0,
            (double) cube4->viewPointXYZ.Z() -
            (double) cube4->datasetSizeXYZ.Z() / 2.0);
        viewVector /= viewVector.Norm();

40
        diffuseIntensity = 0;
        specularIntensity = 0;

        for (int light=0; light<cube4->lightSourceXYZ.Size(); ++light) {

            lightVector = cube4->lightSourceXYZ[light].Direction();
45
            lightVector /= lightVector.Norm();

            //halfway = lightVector + viewVector; // skipp /2.0 because
normalization
            //norm = halfway.Norm();
50
}
}

```

```

294
halfway(0,0,0);

5      double NL = surfaceNormal * lightVector;
lightReflection = surfaceNormal;
lightReflection *= (2*NL);
lightReflection = lightReflection - lightVector;

10     if (NL < 0)
          NL = 0;
else
      NL = pow(NL, cube4->lightSourceXYZ[light].Sharpness());

15     double phongComponent;

//double NH = surfaceNormal*halfway;
//if (NH < 0)
//  phongComponent = 0;
//else
//  phongComponent = pow(NH, cube4->phongExponent);

20     double RV = lightReflection * viewVector;
if (RV < 0)
  phongComponent = 0;
else
  phongComponent = pow(RV, cube4->phongExponent);

25     diffuseIntensity +=
  cube4->lightSourceXYZ[light].Intensity() * NL;
specularIntensity +=
  cube4->lightSourceXYZ[light].Intensity() *

30     phongComponent;
}
}

35     else if (cube4->shaderMode == RefMap) {
Vector3D<FixPointNumber> diff, spec;
Vector3D<FixPointNumber> gradient(-(*inputs.gx),
  -(*inputs gy),
  -(*inputs.gz));

40     reflectanceMap->DiffuseLookUp(gradient,
  cube4->reflectanceMapIP,
  cube4->distortionCompensation,
  diff);

45
50

```

55

```

295
reflectanceMap- >SpecularLookUp(gradient,
5
cube4->reflectanceMapIP,
cube4->distortionCompensation,
spec);

10
diffuseIntensity( diff.R(), diff.G(), diff.B());
specularIntensity( spec.R(), spec.G(), spec.B());
}
else if (cube4->shaderMode == RefVector) {

15
Vector3D<FixPointNumber> diff, spec;
Vector3D<FixPointNumber> gradient(-(*inputs.gx),
-(*inputs gy),
-(*inputs.gz));
20

Vector3D<FixPointNumber> eyeReflection, refNormal, refView;

// viewVector is inverse of sightRay
Vector3D<FixPointNumber> viewVectorFixP(cube4->viewPointXYZ.X()
25
-cube4-
>datasetSizeXYZ.X() / 2.0,
cube4->viewPointXYZ.Y()
30
-cube4-
>datasetSizeXYZ.Y() / 2.0,
cube4->viewPointXYZ.Z()
35
-cube4-
>datasetSizeXYZ.Z() / 2.0;

FixPointNumber gradientDotProduct = gradient * gradient;
FixPointNumber gradientViewVectorDotProduct = gradient *
viewVectorFixP;
40
refNormal = gradient;
refNormal *= (2 * gradientViewVectorDotProduct);
refView = viewVectorFixP;
refView *= gradientDotProduct;
45
eyeReflection = refNormal - refView;
reflectanceMap->DiffuseLookUp(gradient,
cube4->reflectanceMapIP,
50

```

```

5           cube4->distortionCompensation,
diff);

reflectanceMap->SpecularLookUp(eyeReflection,
10          cube4->reflectanceMapIP,
cube4->distortionCompensation,
spec);

15          diffuseIntensity( diff.R(), diff.G(), diff.B());
specularIntensity( spec.R(), spec.G(), spec.B());
}

// The following lines contain the functionality of a color blending unit

20 // adjust the surface illumination to the degree of which a surface
// is present at the current sample position
//diffuseIntensity *= signalToNoiseGradientMagnitude;
//specularIntensity *= signalToNoiseGradientMagnitude;

25 // adjust the surface illumination to the degree specified by Kd and Ks
diffuseIntensity *= cube4->Kd;
specularIntensity *= cube4->Ks;

// add ambient light (surface independent)
diffuseIntensity += cube4->IAmbient * cube4->Ka;

30 // combine color transfer function result with illumination result
if (cube4->shaderMode == NoShading){
    results.shadel->r = red;
    results.shadel->g = green;
    results.shadel->b = blue;
}
35 else{
    results.shadel->r = red * diffuseIntensity.X() +
specularIntensity.X();
    results.shadel->g = green * diffuseIntensity.Y() +
specularIntensity.Y();
40    results.shadel->b = blue * diffuseIntensity.Z() +
specularIntensity.Z();
}

45 // clip too large color intensities
results.shadel->r = (Min(1, results.shadel->r));
results.shadel->g = (Min(1, results.shadel->g));
results.shadel->b = (Min(1, results.shadel->b));

// clip too small color intensities
50 results.shadel->r = (Max(0, results.shadel->r));

```

```

297
results.shadel->g = (Max(0,           results.shadel->g));
results.shadel->b = (Max(0, results.shadel->b));

5   // Levoy classification

//double offset      = ABS(cube4->threshold - intensity);

10  //double currHalfWidth = cube4->maxWidth/2.0 * gradientMagnitude;
//double extHalfWidth  = currHalfWidth + 1.0/255.0;
//double alpha = 0;
//if (offset < currHalfWidth)
//alpha = cube4->mulAlpha * (extHalfWidth - offset) / (extHalfWidth);

15  //if (alpha > 1.0)
//alpha = 1.0;
//results.shadel->a = alpha;

20  // assign alpha
results.shadel->a = double(colorLUT->Alpha(index)/255.0);

25  // premultiply rgb with a
results.shadel->r *= results.shadel->a;
results.shadel->g *= results.shadel->a;
results.shadel->b *= results.shadel->a;

30  if (inputs.perPipelineControlFlags->invalid) {
    results.shadel->r = 0;
    results.shadel->g = 0;
    results.shadel->b = 0;
    results.shadel->a = 0;
}
*/
35  results.shadel->r = inputs.voxel->raw16bit % 32;
    results.shadel->g = inputs.voxel->raw16bit / 32 % 32;
    results.shadel->b = inputs.voxel->raw16bit / 32 / 32 % 32;
// results.shadel->a = inputs.perPipelineControlFlags->voxelPosXYZ.Z();
results.shadel->a = inputs.voxel->raw16bit;

40  } // RunForOneClockCycle

///////////////////////////////
// internal utility functions

45  // end of ShaderPipeline.C
:::::::::::
cube4/ShaderPipeline.h
:::::::::::
50  // ShaderPipeline.h
// (c) Ingmar Bitter '97

```

```

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

5      #ifndef _ShaderPipeline_h_      // prevent multiple includes
# define _ShaderPipeline_h_

10     #include "Misc.h"
# include "Object.h"
# include "Voxel.h"
# include "Shadel.h"
# include "Control.h"
# include "FixPointNumber.h"
15     #include "ReflectanceMap.h"
# include "ColorLUT.h"

20     class ShaderStage;
class Cube4;

25     class ShaderPipelineInputs {
public: // pointers
    Voxel *voxel;
    ScalarGradient *gx, *gy, *gz;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags,
};

30     class ShaderPipelineResults {
public: // pointers
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags,
};

35

40     class ShaderPipeline : virtual public Object {
public:

45         static void Demo ();

        // constructors & destructors
        ShaderPipeline ();
        ~ShaderPipeline ();

        // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream &) const;

50         // - local show/set functions
        static void GlobalSetup (const int setNumOfChips,

```

299

```

    const int setNumberOfPipelinesPerChip);

5     virtual void LocalSetup(const int setChipIndex,
                           const int setPipelineIndex,
                           ShaderStage & shaderStage);
10    virtual void PerFrameSetup();
// local computation functions
    virtual void RunForOneClockCycle();

public:
15    ShaderPipelineInputs inputs;
    ShaderPipelineResults results;

protected:
    static int    numOFChips, numOFpipelinesPerChip;
    static Cube4 *cube4;
20    ColorLUT *colorLUT;
    ReflectanceMap *reflectanceMap;
    int           chipIndex, pipelineIndex; // only for debugging purpose
};

25 #include "ShaderStage.h"
#include "Cube4.h"

#ifndef _ShaderPipeline_h_
::::::::::
cube4/ShaderStage.C
::::::::::
30 // ShaderStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.
35 #include "ShaderStage.h"

void ShaderStage::Demo()
{
    ShaderStage shader;
40    cout << endl << "Demo of class " << typeid(shader).name();
    cout << endl << "size : " << sizeof(ShaderStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "ShaderStage shader; = " << shader;
    cout << endl << "End of demo of class " << typeid(shader).name() << endl;
45 } // Demo

///////////////////////////////
// constructors & destructors
50

```

55

```

300
// static first init
int ShaderStage::numOfChips          = 0;
int ShaderStage::numOfPipelinesPerChip = 0;
Cube4 *ShaderStage::cube4 = 0;

5   ShaderStage::ShaderStage()
{
    shaderPipeline = new ShaderPipeline [numOfPipelinesPerChip];
    results.shadel = new Shadel [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
[numOfPipelinesPerChip];
} // defaultconstructor

10
15
ShaderStage::~ShaderStage()
{
    if (shaderPipeline) { delete shaderPipeline; shaderPipeline=0; }
    if (results.shadel) { delete results.shadel; results.shadel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
20 }
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

25

///////////////////////////////
// show/set data & data properties

30
ostream & ShaderStage::Ostream(ostream & os) const
{
    // append ShaderStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << " numOfChips           = " << numOfChips;
35    os << endl << " numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << " chipIndex            = " << chipIndex;

    // return complete os
    return os;
}

40 } // Ostream

///////////////////////////////
// show/set data & data properties
45 // - local show/set functions

50 void ShaderStage::GlobalSetup(const int setNumOfChips,

```

55

301

```

    const int setNumOfPipelinesPerChip,
5      Cube4 *setCube4)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    ShaderPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip);
10   cube4 = setCube4;
} // GlobalSetup

void ShaderStage::LocalSetup(const int setChipIndex)
15 {
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        shaderPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup

20 void ShaderStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
25       results.shadel[p].r = results.shadel[p].g = results.shadel[p].b = 0;
       results.shadel[p].a = 0;
       results.weightsXYZ[p](0,0,0);
       results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();
30
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        shaderPipeline[p].PerFrameSetup();
    }

    //cout << endl << cube4->shaderMode;

35
    if (cube4->shaderMode == RefMap) {
        // setup parallel viewVector
        Vector3D<double> viewVector;
        viewVector( (double) cube4->viewPointXYZ.X() - (double) cube4-
40 >datasetSizeXYZ.X() / 2.0,
                    (double) cube4->viewPointXYZ.Y()
- (double) cube4->datasetSizeXYZ.Y() / 2.0,
                    (double) cube4->viewPointXYZ.Z() -
(double) cube4->datasetSizeXYZ.Z() / 2.0);
        viewVector /= viewVector.Norm();
45
        // setup reflectance map

        cout << endl << "start computing reflectance table";
    }
}

```

55

```

5      Timer timer;
reflectanceMap.PerFrameSetup(cube4->reflectanceMapBits,
                           cube4->mapWeightBits,
                           cube4->constantAngularResolution,
10     false,
                           viewVector,
                           cube4->lightSourceXYZ,
15     cube4->phongExponent);
cout << endl << "done computing reflectance table " << timer <<
endl;
}

20   else if (cube4->shaderMode == RefVector){

// setup dummy viewVector

        Vector3D<double> viewVector(0, 0, 0);

25   // setup reflectance map
cout << endl << "start computing reflectance table";

        Timer timer;
reflectanceMap.PerFrameSetup(cube4->reflectanceMapBits,
                           cube4->mapWeightBits,
                           cube4->constantAngularResolution,
                           true,
35     viewVector,
                           cube4->lightSourceXYZ,
                           cube4->phongExponent);
cout << endl << "done computing reflectance table " << timer <<
endl;
}

40   // cout << endl << "load lookup table";
colorLUT.ReadTable(cube4->colorTableName);

45   // print debug info
//static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

```

50

```

////////// // local computation functions

5   // local computation functions

void ShaderStage::RunForOneClockCycle()
{
    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        shaderPipeline[p].RunForOneClockCycle();
        results.weightsXYZ[p] = inputs.weightsXYZ[p];
        results.perPipelineControlFlags[p] =
10   inputs.perPipelineControlFlags[p];
    }
    results.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle

////////// // internal utility functions

20

// end of ShaderStage.C
:::::::::::
cube4/ShaderStage.h
25 :::::::::::
// ShaderStage.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30 #ifndef _ShaderStage_h_ // prevent multiple includes
#define _ShaderStage_h_

#include "Misc.h"
#35 include "Object.h"
#include "Voxel.h"
#include "Shadel.h"
#include "Control.h"
#include "ShaderPipeline.h"
#include "FixPointNumber.h"
40 #include "Cube4.h"
#include "ReflectanceMap.h"
#include "ColorLUT.h"

class Cube4;
45

class ShaderStageInputs {
public: // pointers
    Voxel *voxel;
    ScalarGradient *gx, *gy, *gz;
50    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags     *perChipControlFlags;
}

```

```

      PerPipelineControlFlags           *perPipelineControlFlags;
};

5

class ShaderStageResults {
public: // arrays
    Shadel *shadel;
    Vector3D<FixPointNumber> *weightsXYZ;
10   PerChipControlFlags     perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

15 class ShaderStage : virtual public Object {
public:

    static void      Demo ();

20   // constructors & destructors
    ShaderStage ();
    ~ShaderStage ();

    // show/set data & data properties
    // - class Object requirements
25   virtual ostream & Ostream (ostream &) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
30   const int setNumOfPipelinesPerChip,

    Cube4 *setCube4);
    virtual void LocalSetup (const int setChipIndex);
    virtual void PerFrameSetup ();
35   // local computation functions
    virtual void RunForOneClockCycle();

public:
    ShaderPipeline *shaderPipeline;
40   ShaderStageInputs inputs;
    ShaderStageResults results;

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
45   int         chipIndex; // only for debugging purpose
    ReflectanceMap reflectanceMap;
    ColorLUT colorLUT;

    friend class ShaderPipeline;
50   };

#endif // _ShaderStage_h_

```

55

305

```

:::::::::::
cube4/SliceVoxelFiFoPipeline.C
:::::::::::
// SliceVoxelFiFoPipeline.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

10 #include "SliceVoxelFiFoPipeline.h"

void SliceVoxelFiFoPipeline::Demo()
{
    SliceVoxelFiFoPipeline sliceVoxelFiFo;
    cout << endl << "Demo of class " << typeid(sliceVoxelFiFo).name();
    cout << endl << "size : " << sizeof(SliceVoxelFiFoPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "SliceVoxelFiFo pipeline sliceVoxelFiFo; = " <<
sliceVoxelFiFo;
    cout << endl << "End of demo of class " << typeid(sliceVoxelFiFo).name() <<
20 endl;
} // Demo

///////////////////////////////
// constructors & destructors

25 // static first init
int SliceVoxelFiFoPipeline::numOfChips           = 0;
int SliceVoxelFiFoPipeline::numOfPipelinesPerChip = 0;
int SliceVoxelFiFoPipeline::blockSize            = 0;
30 int SliceVoxelFiFoPipeline::maxDatasetSizeX     = 256;
int SliceVoxelFiFoPipeline::maxDatasetSizeY     = 256;
Cube4 *SliceVoxelFiFoPipeline::cube4 = 0;

SliceVoxelFiFoPipeline::SliceVoxelFiFoPipeline()
{
35     // step delay for a z-step within a block
     blockSliceDelay = (maxDatasetSizeX*blockSize
                         ) /
     (numOfChips*numOfPipelinesPerChip);

        // step delay for a z-step between blocks
40     volumeSliceDelay = (maxDatasetSizeX*maxDatasetSizeY
                         ) /
     (numOfChips*numOfPipelinesPerChip);

     blockSliceVoxelFiFo.SetSize(blockSliceDelay);
     blockSliceWeightsFiFo.SetSize(blockSliceDelay);
45     blockSlicePerPipelineControlFlagsFiFo.SetSize(blockSliceDelay);
     blockSlicePerChipControlFlagsFiFo.SetSize(blockSliceDelay);

     volumeSliceVoxelFiFo.SetSize(volumeSliceDelay);
}

```

50

55

306

```

    const int setPipelineIndex,
5   SliceVoxelFiFoStage & sliceVoxelFiFoStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;
    sliceVoxelFiFoStageChip = sliceVoxelFiFoStage.sliceVoxelFiFoStageChip;
10   startReadWriteCounters = sliceVoxelFiFoStage.startReadWriteCounters;

    inputs.voxel = &(sliceVoxelFiFoStage.inputs.voxel[pipelineIndex]);
    inputs.weightsXYZ =
&(sliceVoxelFiFoStage.inputs.weightsXYZ[pipelineIndex]);
15   inputs.perChipControlFlags
        = sliceVoxelFiFoStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        =
&(sliceVoxelFiFoStage.inputs.perPipelineControlFlags[pipelineIndex]);

20   results.voxel = &(sliceVoxelFiFoStage.results.voxel[pipelineIndex]);
    results.weightsXYZ =
&(sliceVoxelFiFoStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
        =
25   &(sliceVoxelFiFoStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(sliceVoxelFiFoStage.results.perChipControlFlags);
} // LocalSetup

30   void SliceVoxelFiFoPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in SliceVoxelFiFoStage

    // resize fifo's according to dataset size
    // step delay for a z-step within a block
35   blockSliceDelay = (cube4->datasetSizeXYZ.X()*blockSize
                                ) /
(numOfChips*numOfPipelinesPerChip);

    // step delay for a z-step between blocks
40   volumeSliceDelay = (cube4->datasetSizeXYZ.X()*cube4->datasetSizeXYZ.Y()
                                ) /
(numOfChips*numOfPipelinesPerChip);

45   blockSliceVoxelFiFo.SetSize(blockSliceDelay);
    blockSliceWeightsFiFo.SetSize(blockSliceDelay);
    blockSlicePerPipelineControlFlagsFiFo.SetSize(blockSliceDelay);
    blockSlicePerChipControlFlagsFiFo.SetSize(blockSliceDelay);

    volumeSliceVoxelFiFo.SetSize(volumeSliceDelay);
    volumeSliceWeightsFiFo.SetSize(volumeSliceDelay);
50

```

```

5           volumeSlicePerPipelineControlFlagsFiFo.SetSize(volumeSliceDelay);
5           volumeSlicePerChipControlFlagsFiFo.SetSize(volumeSliceDelay);

10          blockSliceVoxelFiFo.Preset(*(inputs.voxel));
10          blockSliceWeightsFiFo.Preset(*(inputs.weightsXYZ));
10          blockSlicePerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFl
ags));
10          blockSlicePerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

15          volumeSliceVoxelFiFo.Preset(*(inputs.voxel));
15          volumeSliceWeightsFiFo.Preset(*(inputs.weightsXYZ));
15          volumeSlicePerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlF
lags));
15          volumeSlicePerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

20          readBigFiFoCounter = readSmallFiFoCounter =
20          writeBigFiFoCounter = writeSmallFiFoCounter = -1;
} // PerFrameSetup

25 /////////////////
// local computation functions

void SliceVoxelFiFoPipeline::RunForOneClockCycle()
{
25     // reset counters
25     if (*startReadWriteCounters ||
25         ((readBigFiFoCounter == 0) &&
25          (readSmallFiFoCounter == 0) &&
25          (writeBigFiFoCounter == 0) &&
25          (writeSmallFiFoCounter == 0))) {
30
30         readBigFiFoCounter = blockSliceDelay;
30         readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;

35         writeBigFiFoCounter = readBigFiFoCounter;
35         writeSmallFiFoCounter = readSmallFiFoCounter;
35     }

35 ///////////////
// first read from Fifos into results register

40     // at start of block read from big Fifo
40     if (readBigFiFoCounter > 0) {
40         volumeSliceVoxelFiFo.Read(*(results.voxel));
40         volumeSliceWeightsFiFo.Read(*(results.weightsXYZ));

45         volumeSlicePerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFl
ags));
45         volumeSlicePerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
45         --readBigFiFoCounter;
}

```

50

55

```

309
    // if (this == &cube4-
>sliceVoxelFiFo1[0].sliceVoxelFiFoPipeline[0])
    cout<<"R"<<*(results.voxel)<<volumeSliceVoxelFiFo<<endl;
5
}

// in middle and at end of block read from small FiFo
else if (readSmallFiFoCounter > 0) {
    blockSliceVoxelFiFo.Read(*(results.voxel));
    blockSliceWeightsFiFo.Read(*(results.weightsXYZ));
10

blockSlicePerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFla
gs));

blockSlicePerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
15
    --readSmallFiFoCounter;
    // if (this == &cube4->sliceVoxelFiFo0[0].sliceVoxelFiFoPipeline[0])
cout<<"r";
}

20
///////////////
// now write to FiFos from inputs register

// at start and in middle of block write to small FiFo
if (writeSmallFiFoCounter > 0) {
    blockSliceVoxelFiFo.Write( *(inputs.voxel) );
    blockSliceWeightsFiFo.Write( *(inputs.weightsXYZ) );
25

    blockSlicePerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFla
gs));
    blockSlicePerChipControlFlagsFiFo.Write(
*(inputs.perChipControlFlags));
30
    --writeSmallFiFoCounter;
    // if (this == &cube4->sliceVoxelFiFo0[0].sliceVoxelFiFoPipeline[0])
cout<<"w";
}

35
// at end of block write to big FiFo of next chip
else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOfChips);
    int p(pipelineIndex);
    SliceVoxelFiFoPipeline
        *next(&sliceVoxelFiFoStageChip[c].sliceVoxelFiFoPipeline[p]);
    next->volumeSliceVoxelFiFo.Write( *(inputs.voxel) );
    next->volumeSliceWeightsFiFo.Write( *(inputs.weightsXYZ) );
    next-
40
>volumeSlicePerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags))
;
    next-
>volumeSlicePerChipControlFlagsFiFo.Write(*(inputs.perChipControlFlags));
    --writeBigFiFoCounter;
    //if (this == &cube4->sliceVoxelFiFo1[0].sliceVoxelFiFoPipeline[0])
cout<<"W";
45

50

```

55

310

}

5 } // RunForOneClockCycle

```
//////////////////////////////  
// internal utility functions
```

10

```
// end of SliceVoxelFiFoPipeline.C
```

```
::::::::::::
```

```
cube4/SliceVoxelFiFoPipeline.h
```

```
::::::::::::
```

15

```
// SliceVoxelFiFoPipeline.h
```

```
// (c) Ingmar Bitter '97
```

```
// Copyright, Mitsubishi Electric Information Technology Center  
// America, Inc., 1997, All rights reserved.
```

20

```
#ifndef _SliceVoxelFiFoPipeline_h_ // prevent multiple includes  
#define _SliceVoxelFiFoPipeline_h_
```

30

```
#include "Misc.h"  
#include "Object.h"  
#include "FiFo.h"  
#include "Voxel.h"  
#include "Coxel.h"  
#include "Control.h"  
#include "FixPointNumber.h"
```

35

```
typedef Vector3D<FixPointNumber> FixPointVector3D;
```

```
class SliceVoxelFiFoStage;  
class Cube4;
```

40

```
class SliceVoxelFiFoPipelineInputs {  
public: // pointers  
    Voxel *voxel;  
    Vector3D<FixPointNumber> *weightsXYZ;  
    PerChipControlFlags *perChipControlFlags;  
    PerPipelineControlFlags *perPipelineControlFlags;  
};
```

45

```
class SliceVoxelFiFoPipelineResults {  
public: // pointers  
    Voxel *voxel;  
    Vector3D<FixPointNumber> *weightsXYZ;  
    PerChipControlFlags *perChipControlFlags;  
    PerPipelineControlFlags *perPipelineControlFlags;  
};
```

55

```

class SliceVoxelFiFoPipeline : virtual public Object {
5   public:

    static void      Demo ();

    // constructors & destructors
    SliceVoxelFiFoPipeline ();
10   ~SliceVoxelFiFoPipeline ();

    // show/set data & data properties
    // - class Object requirements
    virtual ostream & Ostream (ostream & ) const;
15

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
20                           const int setBlockSize,
                           Cube4 *setCube4);

    virtual void LocalSetup(const int setChipIndex,
                           const int setPipelineIndex,
                           SliceVoxelFiFoStage & sliceVoxelFiFoStage);
                           virtual void PerFrameSetup();
30   // local computation functions
                           virtual void RunForOneClockCycle();

    public:
        SliceVoxelFiFoPipelineInputs inputs;
35   SliceVoxelFiFoPipelineResults results;

    protected:
        bool *startReadWriteCounters;
        Fifo<Voxel>           blockSliceVoxelFiFo;
40       Fifo<FixPointVector3D> blockSliceWeightsFiFo;
        Fifo<PerPipelineControlFlags> blockSlicePerPipelineControlFlagsFiFo;
        Fifo<PerChipControlFlags>  blockSlicePerChipControlFlagsFiFo;

        Fifo<Voxel>           volumeSliceVoxelFiFo;
        Fifo<FixPointVector3D> volumeSliceWeightsFiFo;
45       Fifo<PerPipelineControlFlags> volumeSlicePerPipelineControlFlagsFiFo;
        Fifo<PerChipControlFlags>  volumeSlicePerChipControlFlagsFiFo;

        int blockSliceDelay, volumeSliceDelay;

50       int  readSmallFiFoCounter, readBigFiFoCounter;
        int  writeSmallFiFoCounter, writeBigFiFoCounter;

```

```

312
      SliceVoxelFiFoStage           *sliceVoxelFiFoStageChip; // to access
neighbor chips
5       static int    numChips, numPipelinesPerChip, blockSize;
       static int    maxDatasetSizeX,maxDatasetSizeY;
       static Cube4  *cube4;
       int          chipIndex, pipelineIndex;

       friend class Cube4;
10      }; // class SliceVoxelFiFoPipeline

#include "SliceVoxelFiFoStage.h"
#include "Cube4.h"

#ifndef      // _SliceVoxelFiFoPipeline_h_
15      ::::::::::::::
cube4/SliceVoxelFiFoStage.C
::::::::::::
// SliceVoxelFiFoStage.C
// (c) Ingmar Bitter '97

20      // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "SliceVoxelFiFoStage.h"

void SliceVoxelFiFoStage::Demo()
25      {
        SliceVoxelFiFoStage sliceVoxelFiFo;
        cout << endl << "Demo of class " << typeid(sliceVoxelFiFo).name();
        cout << endl << "size : " << sizeof(SliceVoxelFiFoStage) << " Bytes";
        cout << endl << "public member functions:";
        cout << endl << "SliceVoxelFiFoStage sliceVoxelFiFo; = " << sliceVoxelFiFo;
30      cout << endl << "End of demo of class " << typeid(sliceVoxelFiFo).name() <<
endl;
} // Demo

35      /////////////////////////////////
// constructors & destructors

// static first init
int SliceVoxelFiFoStage::numChips           = 0;
int SliceVoxelFiFoStage::numPipelinesPerChip = 0;
int SliceVoxelFiFoStage::blockSize           = 0;
40      Cube4 *SliceVoxelFiFoStage::cube4 = 0;

SliceVoxelFiFoStage::SliceVoxelFiFoStage()
{
    sliceVoxelFiFoPipeline = new SliceVoxelFiFoPipeline
45      [numPipelinesPerChip];
    results.voxel = new Voxel [numPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numPipelinesPerChip];
}

```

50

55

```

      results.perPipelineControlFlags = new PerPipelineControlFlags
313
[numberOfPipelinesPerChip];
} // defaultconstructor

5

SliceVoxelFiFoStage::~SliceVoxelFiFoStage()
{
    if (sliceVoxelFiFoPipeline) { delete sliceVoxelFiFoPipeline;
10   sliceVoxelFiFoPipeline=0; }
    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
15   results.perPipelineControlFlags=0;
    }
} // destructor

20 /////////////////
// show/set data & data properties

ostream & SliceVoxelFiFoStage::Ostream(ostream & os) const
{
25   // append SliceVoxelFiFoStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips          = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex           = " << chipIndex;

30   // return complete os
    return os;
}

} // Ostream

35 /////////////////
// show/set data & data properties
//
// - local show/set functions

40 void SliceVoxelFiFoStage::GlobalSetup(const int setNumOfChips,
                                         const int setNumOfPipelinesPerChip,
                                         const int setBlockSize,
                                         Cube4 *setCube4)
45
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
}

```

50

55

```

314
blockSize          =      setBlockSize;
cube4 = setCube4;
SliceVoxelFiFoPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
5                                     blockSize, cube4);

} // GlobalSetup

10 void SliceVoxelFiFoStage::LocalSetup(const int setChipIndex,
SliceVoxelFiFoStage *setSliceVoxelFiFoStageChip)
{
    chipIndex = setChipIndex;
    sliceVoxelFiFoStageChip = setSliceVoxelFiFoStageChip;
15   for (int p=0; p<numOfPipelinesPerChip; ++p) {
        sliceVoxelFiFoPipeline[p].LocalSetup(chipIndex,p,*this);
    }
} // LocalSetup

20 void SliceVoxelFiFoStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    for (p=0; p<numOfPipelinesPerChip; ++p) {
25      results.voxel[p].raw16bit = 0;
      results.weightsXYZ[p](0,0,0);
      results.perPipelineControlFlags.Reset();
    }
    results.perChipControlFlags.Reset();

30   for (p=0; p<numOfPipelinesPerChip; ++p) {
        sliceVoxelFiFoPipeline[p].PerFrameSetup();
    }

    // print debug info
35   //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

40 ///////////////////////////////////////////////////////////////////
// local computation functions
void SliceVoxelFiFoStage::RunForOneClockCycle()
{
    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
45      sliceVoxelFiFoPipeline[p].RunForOneClockCycle();
    }
} // RunForOneClockCycle

50 ///////////////////////////////////////////////////////////////////

```

315

```

// internal utility functions

5
// end of SliceVoxelFiFoStage.C
:::::::::::
cube4/Test.C
:::::::::::
10 // Test.C
// (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15 #include "Test.h"

void Test::Run(int argc, char *argv[])
{
20     Cube4 cube4(argc, argv);
     cube4.RenderAnimation();
} // Run

// end of Test.C
:::::::::::
25 cube4/SliceVoxelFiFoStage.h
:::::::::::
// SliceVoxelFiFoStage.h
// (c) Ingmar Bitter '97

30 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#ifndef _SliceVoxelFiFoStage_h_      // prevent multiple includes
#define _SliceVoxelFiFoStage_h_
35

#include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Control.h"
40 #include "SliceVoxelFiFoPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"

class Cube4;

45 class SliceVoxelFiFoStageInputs {
public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
50    PerPipelineControlFlags *perPipelineControlFlags;
};


```

55

316

```

class SliceVoxelFiFoStageResults {
public: // arrays
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags     perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

10   class SliceVoxelFiFoStage : virtual public Object {
public:

    static void      Demo ();

15    // constructors & destructors
    SliceVoxelFiFoStage ();
    ~SliceVoxelFiFoStage ();

    // show/set data & data properties
    // - class Object requirements
20    virtual ostream & Ostream (ostream &) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
25    const int setNumOfPipelinesPerChip,
    const int setBlockSize,
    Cube4 *setCube4);
    virtual void LocalSetup (const int setChipIndex,
30    SliceVoxelFiFoStage *setSliceVoxelFiFoStageArray);
    virtual void PerFrameSetup ();
    // local computation functions
    virtual void RunForOneClockCycle();

35   public:
    SliceVoxelFiFoPipeline *sliceVoxelFiFoPipeline;
    SliceVoxelFiFoStageInputs inputs;
    SliceVoxelFiFoStageResults results;
    bool *startReadWriteCounters;

40   protected:
    SliceVoxelFiFoStage *sliceVoxelFiFoStageChip; // to access neighbor chips
    static int   numOfChips, numOfPipelinesPerChip, blockSize;
    static Cube4 *cube4;
    int         chipIndex; // only for debugging purpose

45   friend class Cube4;
   friend class SliceVoxelFiFoPipeline;
};

50

```

55

317

```

#endif // _SliceVoxelFiFoStage_h_
:::::::::::
5   cube4/Test.h
:::::::::::
// Test.h
// (c) Ingmar Bitter '97

10  // Copyright, Mitsubishi Electric Information Technology Center
    // America, Inc., 1997, All rights reserved.

#ifndef _Test_h_           // prevent multiple includes
#define _Test_h_

15  #include "Cube4.h"

    class Test { public: static void Run(int argc, char *argv[]); };

#endif // _Test_h_
:::::::::::
20  cube4/TriLinXPipeline.C
:::::::::::
// TriLinXPipeline.C
// (c) Ingmar Bitter '97

25  // Copyright, Mitsubishi Electric Information Technology Center
    // America, Inc., 1997, All rights reserved.

#include "TriLinXPipeline.h"

30  void TriLinXPipeline::Demo()
{
    TriLinXPipeline triLinX;
    cout << endl << "Demo of class " << typeid(triLinX).name();
    cout << endl << "size : " << sizeof(TriLinXPipeline) << " Bytes";
    cout << endl << "public member functions:";

35  cout << endl << "TriLinXPipeline triLinX;" << triLinX;
    cout << endl << "End of demo of class " << typeid(triLinX).name() << endl;
} // Demo

40  /////////////////////////////////
// constructors & destructors

    // static first init
    int TriLinXPipeline::numOfChips          = 0;
    int TriLinXPipeline::numOfPipelinesPerChip = 0;
45  int TriLinXPipeline::blockSize          = 0;
    Cube4 *TriLinXPipeline::cube4            = 0;

    TriLinXPipeline::TriLinXPipeline()
    {
50      // Number of clock cycles to process a partial beam inside a block
        int partialBeamDelay = blockSize / numOfPipelinesPerChip;
}

```

55

318

```

    // Delay for communication: 16 bits, 4 bits / cycle, double frequency => 2
5   cycles
    int communicationDelay = 2;

    partialBeamVoxelFiFo.SetSize(partialBeamDelay);
    partialBeamWeightsFiFo.SetSize(partialBeamDelay);
    partialBeamPerPipelineControlFlagsFiFo.SetSize(partialBeamDelay);
10  partialBeamPerChipControlFlagsFiFo.SetSize(partialBeamDelay);

    communicationDelayVoxelFiFo.SetSize(communicationDelay);
    communicationDelayWeightsFiFo.SetSize(communicationDelay);
    communicationDelayPerPipelineControlFlagsFiFo.SetSize(communicationDelay);
15  communicationDelayPerChipControlFlagsFiFo.SetSize(communicationDelay);

} // constructor

TriLinXPipeline::~TriLinXPipeline()
{
20 } // destructor

////////////////////////////////////////////////////////////////
// show/set data & data properties

25 ostream & TriLinXPipeline::Ostream(ostream & os) const
{
    // append TriLinXPipeline info to os
    os << typeid(*this).name() << " " << (void *) this;
30    os << endl << " numOfChips      = " << numOfChips;
    os << endl << " numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << " chipIndex        = " << chipIndex;

    // return complete os
    return os;
35
} // Ostream

////////////////////////////////////////////////////////////////
40 // show/set data & data properties
//
// - local show/set functions

void TriLinXPipeline::GlobalSetup(const int setNumOfChips,
45          const int setNumOfPipelinesPerChip,
          const int setBlockSize)
{
    numOfChips      = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    blockSize       = setBlockSize;
50
}

```

55

319

```

} // GlobalSetup

5   void TriLinXPipeline::LocalSetup(const int setChipIndex,
                                    const int setPipelineIndex,
                                    TriLinXStage & triLinXStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.voxelA = &(triLinXStage.computation.voxel[pipelineIndex]);
    inputs.voxelB = &(triLinXStage.computation.voxel[pipelineIndex+1]);
    inputs.weightsXYZ = &(triLinXStage.computation.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
        = &(triLinXStage.computation.perChipControlFlags);
    inputs.perPipelineControlFlags
        = &(triLinXStage.computation.perPipelineControlFlags[pipelineIndex]);

    results.voxel = &(triLinXStage.results.voxel[pipelineIndex]);
    results.weightsXYZ = &(triLinXStage.results.weightsXYZ[pipelineIndex]);
20   results.perPipelineControlFlags
        = &(triLinXStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(triLinXStage.results.perChipControlFlags);

    cube4 = triLinXStage(cube4;
25   } // LocalSetup

void TriLinXPipeline::PerFrameSetup()
{
30   // reset pipeline registers already done in TriLinXStage
    partialBeamVoxelFiFo.Preset(*(inputs.voxelA));
    partialBeamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    partialBeamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
    partialBeamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

35   communicationDelayVoxelFiFo.Preset(*(inputs.voxelA));
    communicationDelayWeightsFiFo.Preset(*(inputs.weightsXYZ));
    communicationDelayPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
    communicationDelayPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));
40   args));
} // PerFrameSetup

45   /////////////////////////////////
// local computation functions
void TriLinXPipeline::RunForOneClockCycle()
{
    // Linear interpolation
}

50

```

55

320

```

// c = w(b-a)+a
//
5 // w=0 -> c=a
// w=1 -> c=b

if (cube4->cubeMode == Cube4Light) {
    // just pick current voxel (as if w=0)
10   results.voxel->raw16bit = inputs.voxelA->raw16bit; // w=0;
    // use weights later on in composing
}

else if (cube4->cubeMode == Cube4Classic) {
    // do real interpolation
15   a((int)inputs.voxelA->raw16bit);
   b((int)inputs.voxelB->raw16bit);
   results.voxel->raw16bit = (short)(inputs.weightsXYZ->X()*(b-a) + a);
    // assign control bits to weights to flow down the rest of the
    // pipeline as weights
20   /**(results.weightsXYZ) = *(inputs.weightsXYZ);
   //results.weightsXYZ->SetX(-1 * StepDirection[inputs.perChipControlFlags-
>xStep]);
}
} // RunForOneClockCycle

25

///////////////////////////////
// internal utility functions

30 // end of TriLinXPipeline.C
:::::::::::
cube4/TriLinXPipeline.h
:::::::::::
// TriLinXPipeline.h
// (c) Ingmar Bitter '97
35

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

40 #ifndef _TriLinXPipeline_h_ // prevent multiple includes
#define _TriLinXPipeline_h_

45 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "FiFo.h"
#include "Coxel.h"
#include "Control.h"
#include "FixPointNumber.h"

50 typedef Vector3D<FixPointNumber> FixPointVector3D;

```

55

321

```

5      class TriLinXStage;
6      class Cube4;
7
8      class TriLinXPipelineInputs {
9      public: // pointers
10         Voxel *voxelA;
11         Voxel *voxelB;
12         Vector3D<FixPointNumber> *weightsXYZ;
13         PerChipControlFlags     *perChipControlFlags;
14         PerPipelineControlFlags *perPipelineControlFlags;
15     };
16
17
18     class TriLinXPipelineResults {
19     public: // pointers
20         Voxel *voxel;
21         Vector3D<FixPointNumber> *weightsXYZ;
22         PerChipControlFlags     *perChipControlFlags;
23         PerPipelineControlFlags *perPipelineControlFlags;
24     };
25
26
27     class TriLinXPipeline : virtual public Object {
28     public:
29
30         static void      Demo ();
31
32         // constructors & destructors
33         TriLinXPipeline ();
34         ~TriLinXPipeline ();
35
36         // show/set data & data properties
37         // - class Object requirements
38         virtual ostream & Ostream (ostream & ) const;
39
40         // - local show/set functions
41         static void GlobalSetup (const int setNumOfChips,
42                             const int setNumOfPipelinesPerChip,
43                             const int setBlockSize);
44
45         virtual void LocalSetup(const int setChipIndex,
46                             const int setPipelineIndex,
47                             TriLinXStage & triLinXStage);
48         virtual void PerFrameSetup();
49         // local computation functions
50         virtual void RunForOneClockCycle();
51
52     public:
53         TriLinXPipelineInputs inputs;
54         TriLinXPipelineResults results;
55
56     protected:
57         FixPointNumber           a, b;
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
678
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
846
847
848
848
849
849
850
851
852
853
854
855
856
856
857
858
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
875
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634

```

```

322
5   FIFO<Voxel>           partialBeamVoxelFiFo;
FIFO<FixPointVector3D>    partialBeamWeightsFiFo;
FIFO<PerPipelineControlFlags> partialBeamPerPipelineControlFlagsFiFo;
FIFO<PerChipControlFlags>  partialBeamPerChipControlFlagsFiFo;

10  FIFO<Voxel>           communicationDelayVoxelFiFo;
FIFO<FixPointVector3D>    communicationDelayWeightsFiFo;
FIFO<PerPipelineControlFlags> communicationDelayPerPipelineControlFlagsFiFo;
FIFO<PerChipControlFlags>  communicationDelayPerChipControlFlagsFiFo;

15  static int   numOfChips, numOfPipelinesPerChip, blockSize;
static Cube4 *cube4;
int      chipIndex, pipelineIndex;

19  friend class TriLinXStage;
friend class Cube4;
};

20  #include "TriLinXStage.h"
#include "Cube4.h"

25  #endif      // _TriLinXPipeline_h_
:::::::::::
cube4/TriLinXStage.C
:::::::::::
// TriLinXStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

30  #include "TriLinXStage.h"

void TriLinXStage::Demo()
{
    TriLinXStage triLinX;
    cout << endl << "Demo of class " << typeid(triLinX).name();
    cout << endl << "size : " << sizeof(TriLinXStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "TriLinXStage triLinX; = " << triLinX;
    cout << endl << "End of demo of class " << typeid(triLinX).name() << endl;
} // Demo

40  /////////////////////////////////
// constructors & destructors

45  // static first init
int TriLinXStage::numOfChips        = 0;
int TriLinXStage::numOfPipelinesPerChip = 0;
Cube4 *TriLinXStage::cube4 = 0;

50  TriLinXStage::TriLinXStage()

```

55

323

```

{
    triLinXPipeline = new TriLinXPipeline [numOfPipelinesPerChip];

5      computation.voxel = new Voxel [numOfPipelinesPerChip+1];
    computation.weightsXYZ = new Vector3D<FixPointNumber>
    [numOfPipelinesPerChip];
    computation.perPipelineControlFlags = new PerPipelineControlFlags
    [numOfPipelinesPerChip+1]; // +1 just for debugging

10     results.voxel = new Voxel [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
    [numOfPipelinesPerChip];
} // defaultconstructor

15

TriLinXStage::~TriLinXStage()
{
    if (triLinXPipeline) { delete triLinXPipeline; triLinXPipeline=0; }

20    if (computation.voxel) { delete computation.voxel; computation.voxel=0; }
    if (computation.weightsXYZ) { delete computation.weightsXYZ;
computation.weightsXYZ=0; }
    if (computation.perPipelineControlFlags) {
        delete computation.perPipelineControlFlags;
        computation.perPipelineControlFlags=0;
25    }

    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
30    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

35 /////////////////////////////////
// show/set data & data properties

ostream & TriLinXStage::Ostream(ostream & os) const
{
40    // append TriLinXStage info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << "  numOfChips      = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex       = " << chipIndex;

45    // return complete os
    return os;
}

50

```

55

```

} // Ostream

5   /////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

10 void TriLinXStage::GlobalSetup(const int setNumOfChips,
                                const int setNumOfPipelinesPerChip,
                                Cube4 *setCube4)

15 {
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
    cube4               = setCube4;
    TriLinXPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip, cube4-
20 >blockSize);
} // GlobalSetup

25 void TriLinXStage::LocalSetup(const int setChipIndex)
{
    chipIndex(setChipIndex, numOfChips);
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        triLinXPipeline[p].LocalSetup(chipIndex, p, *this);
    }
} // LocalSetup

30

void TriLinXStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
    readBufferVoxel.raw16bit = 0;
    communicationVoxel.raw16bit = 0;
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        computation.voxel[p].raw16bit = 0;
        computation.weightsXYZ[p](0,0,0);
        computation.perPipelineControlFlags[p].Reset();
    }
    computation.voxel[numOfPipelinesPerChip].raw16bit = 0;

    35   for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

    40   for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

    45   for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

    50   for (p=0; p<numOfPipelinesPerChip; ++p) {

```

```

    triLinXPipeline[p].PerFrameSetup();
}
5
    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

10
///////////
// local computation functions

void TriLinXStage::CommunicateForOneClockCycle()
{
15
    ///////////
    // fill communication register

    // only sent data at beginning of a block
    if (inputs.perChipControlFlags->xBlockStart) {

20
        // write readbuffer to communication register
        cube4->triLinX[chipIndex-1].communicationVoxel =
            cube4->triLinX[chipIndex-1].readBufferVoxel;
        cube4->triLinX[chipIndex-1].communicationPerPipelineControlFlags =
            cube4->triLinX[chipIndex-1].readBufferPerPipelineControlFlags;

25
        // leftmost chip sends data as early as possible
        if (inputs.perChipControlFlags->leftmostChip) {
            cube4->triLinX[chipIndex-1].readBufferVoxel =
                inputs.voxel[0];
            cube4->triLinX[chipIndex-1].readBufferPerPipelineControlFlags =
                inputs.perPipelineControlFlags[0];
30
        }
        // remaining chips send (blockSize/numOfPipelinesPerChip) later
        else {
            cube4->triLinX[chipIndex-1].readBufferVoxel =
                triLinXPipeline[0].partialBeamVoxelFiFo.Peek(0);
            cube4->triLinX[chipIndex-1].readBufferPerPipelineControlFlags =
                triLinXPipeline[0].partialBeamPerPipelineControlFlagsFiFo.Peek(0);
35
        }
    }
40
} // CommunicateForOneClockCycle

void TriLinXStage::RunForOneClockCycle()
{
    /*
45
    +---+-----+---+           +---+-----+---+
    | 8 | 8 | 8 | 8 |           | 8 | 8 | 8 | 8 |      inputs
    +---+-----+---+           +---+-----+---+
                           +---+     |           +---+   read

```

50

55

```

      | 5 |<---|?|<-+ 326
      +---+ +--+
      | 3 | ^ | 5 | buffer
      +---+ | 3 | communication
           +---+ register
10   +---+-----+-----+
      | 7 | 7 | 7 | 7 |
      +---+-----+-----+
      | 6 | 6 | 6 | 6 |
      +---+-----+-----+
15   +---+-----+-----+
      | 5 | 5 | 5 | 5 |
      +---+-----+-----+
      | 4 | 4 | 4 | 4 |
      +---+-----+-----+ V
      +---+-----+-----+ >|?|
      +---+-----+-----+ >|?|
      +---+-----+-----+ V
      +---+-----+-----+
      | 3 | 3 | 3 | 3 | 4 |
      +---+-----+-----+ computation
                           register
20
25   +---+-----+-----+
      | 2 | 2 | 2 | 2 |
      +---+-----+-----+ results
      */
int p;

30   /////////////////
// computation
results.perChipControlFlags = computation.perChipControlFlags;
for (p=0; p<numOfPipelinesPerChip; ++p) {
    results.weightsXYZ[p] = computation.weightsXYZ[p];
    results.perPipelineControlFlags[p] = computation.perPipelineControlFlags[p];
    triLinXPipeline[p].RunForOneClockCycle();
}

35   /////////////////
// fill computation registers

40   // for rightmost computation register choose between
    // communicationDelayVoxelFiFo and communication register
if (triLinXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Peek(1).xBlockStart)
45   {
    // at end of block copy from right chip
    computation.voxel[numOfPipelinesPerChip]
        = communicationVoxel;
    computation.perPipelineControlFlags[numOfPipelinesPerChip]
}
50

```

```

      =
      327
      communicationPerPipelineControlFlags;
}
else {
    // within block copy from leftmost pipeline
    computation.voxel[numOfPipelinesPerChip]
        = triLinXPipeline[0].communicationDelayVoxelFiFo.Peek(1);
    computation.perPipelineControlFlags[numOfPipelinesPerChip]
        =
10   triLinXPipeline[0].communicationDelayPerPipelineControlFlagsFiFo.Peek(1);
}

// always copy communicationDelayFiFo to remaining computation registers
for (p=0; p<numOfPipelinesPerChip; ++p) {
    computation.voxel[p]
        =
15   triLinXPipeline[p].communicationDelayVoxelFiFo.Read();
    computation.weightsXYZ(p)
        =
        triLinXPipeline[p].communicationDelayWeightsFiFo.Read();
    computation.perPipelineControlFlags[p]
        =
20   triLinXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Read();
}
computation.perChipControlFlags
    =
triLinXPipeline[0].communicationDelayPerChipControlFlagsFiFo.Read();

25   ///////////////////////////////
// Write to communication delay FiFo
for (p=0; p<numOfPipelinesPerChip; ++p) {
    triLinXPipeline[p].communicationDelayVoxelFiFo.Write
        (triLinXPipeline[p].partialBeamVoxelFiFo.Read());
    triLinXPipeline[p].communicationDelayWeightsFiFo.Write
        (triLinXPipeline[p].partialBeamWeightsFiFo.Read());
30   triLinXPipeline[p].communicationDelayPerPipelineControlFlagsFiFo.Write
        (triLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Read());
}
    triLinkPipeline[0].communicationDelayPerChipControlFlagsFiFo.Write
        (triLinkPipeline[0].partialBeamPerChipControlFlagsFiFo.Read());

35   ///////////////////////////////
// Write to partial beam FiFo
for (p=0; p<numOfPipelinesPerChip; ++p) {
    triLinXPipeline[p].partialBeamVoxelFiFo.Write
        (inputs.voxel[p]);
    triLinXPipeline[p].partialBeamWeightsFiFo.Write
        (inputs.weightsXYZ(p));
    triLinXPipeline[p].partialBeamPerPipelineControlFlagsFiFo.Write
        (inputs.perPipelineControlFlags[p]);
}
    triLinkPipeline[0].partialBeamPerChipControlFlagsFiFo.Write
40   (*inputs.perChipControlFlags);
;
// RunForOneClockCycle

```

50

55

```

///////////////////////////////  

// internal utility functions  

5

// end of TriLinXStage.C
:::::::  

cube4/TriLinXStage.h
10 :::::::::::::  

// TriLinXStage.h
// (c) Ingmar Bitter '97
  

15 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.
  

#ifndef _TriLinXStage_h_      // prevent multiple includes.
#define _TriLinXStage_h_
  

20 #include "Misc.h"
#include "Object.h"
#include "ModInt.h"
#include "Voxel.h"
#include "Control.h"
25 #include "TriLinXPipeline.h"
#include "FixPointNumber.h"
#include "Cube4.h"
  

class Cube4;
30
class TriLinXStageInputs {
public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
35    PerChipControlFlags    *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

40 class TriLinXStageResults {
public: // arrays
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags    perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
45};

class TriLinXStage : virtual public Object {
public:
50    static void      Demo ();

```

329

```

// constructors & destructors
TriLinXStage ();
~TriLinXStage ();

// show/set data & data properties
// - class Object requirements
virtual ostream & Ostream (ostream & ) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
                        const int setNumOfPipelinesPerChip,
                        Cube4 *setCube4);
virtual void LocalSetup (const int setChipIndex);
virtual void PerFrameSetup ();
// local computation functions
virtual void CommunicateForOneClockCycle();
virtual void RunForOneClockCycle();

public:
    TriLinXPipeline *triLinXPipeline;
    TriLinXStageInputs inputs;
    TriLinXStageResults computation;
    TriLinXStageResults results;

protected:
    static int    numOfChips, numOfPipelinesPerChip;
    static Cube4 *cube4;
    ModInt        chipIndex;
    Voxel         readBufferVoxel;
    PerPipelineControlFlags readBufferPerPipelineControlFlags; // only for
30   debugging
    Voxel         communicationVoxel;
    PerPipelineControlFlags communicationPerPipelineControlFlags; // only for
debugging

friend class TriLinXPipeline;
friend class Cube4;
};

#endif // _TriLinXStage_h_
:::::::::::
40 cube4/TriLinYPipeline.C
:::::::::::
// TriLinYPipeline.C
// (c) Ingmar Bitter '97

45 // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "TriLinYPipeline.h"

50 void TriLinYPipeline::Demo()
{

```

55

```

      330
  TriLinYPipeline triLinY;
  cout << endl << "Demo of class " << typeid(triLinY).name();
  cout << endl << "size : " << sizeof(TriLinYPipeline) << " Bytes";
  cout << endl << "public member functions:";
  cout << endl << "TriLinYPipeline triLinY; = " << triLinY;
  cout << endl << "End of demo of class " << typeid(triLinY).name() << endl;
} // Demo

10
///////////
// constructors & destructors

// static first init
int TriLinYPipeline::numOfChips          = 0;
15 int TriLinYPipeline::numOfPipelinesPerChip = 0;
int TriLinYPipeline::blockSize           = 0;
int TriLinYPipeline::maxDatasetSizeX     = 256;
Cube4 *TriLinYPipeline::cube4            = 0;

20 TriLinYPipeline::TriLinYPipeline()
{
    // step delay for a y-step within a block
    int beamDelay = (maxDatasetSizeX
                      ) /
    (numOfChips*numOfPipelinesPerChip);

25    // step delay for a y-step between blocks
    int blockBeamDelay = beamDelay * blockSize;

    beamVoxelFiFo.SetSize(beamDelay);
    beamWeightsFiFo.SetSize(beamDelay);
30    beamPerPipelineControlFlagsFiFo.SetSize(beamDelay);
    beamPerChipControlFlagsFiFo.SetSize(beamDelay);

    blockBeamVoxelFiFo.SetSize(blockBeamDelay);
    blockBeamWeightsFiFo.SetSize(blockBeamDelay);
    blockBeamPerPipelineControlFlagsFiFo.SetSize(blockBeamDelay);
35    blockBeamPerChipControlFlagsFiFo.SetSize(blockBeamDelay);
} // constructor

40 TriLinYPipeline::~TriLinYPipeline()
{
} // destructor

45
///////////
// show/set data & data properties

ostream & TriLinYPipeline::Ostream(ostream & os) const
{
    // append TriLinYPipeline info to os
50

```

```

331
os << typeid(*this).name() << "@" << (void *) this;
os << endl << "  numOfChips      = " << numOfChips;
os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
os << endl << "  chipIndex        = " << chipIndex;

5   // return complete os
    return os;

10  } // Ostream

///////////////////////////////
// show/set data & data properties
//
15  // - local show/set functions

void TriLinYPipeline::GlobalSetup(const int setNumOfChips,
20
                                const int setNumOfPipelinesPerChip,
                                const int setBlockSize)
{
    numOfChips          = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
25    blockSize           = setBlockSize;
} // GlobalSetup

void TriLinYPipeline::LocalSetup(const int setChipIndex,
30
                                const int setPipelineIndex,
                                TriLinYStage & triLinYStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;

    inputs.voxel = &(triLinYStage.inputs.voxel[pipelineIndex]);
    inputs.weightsXYZ = &(triLinYStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
40    = triLinYStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags
        = &(triLinYStage.inputs.perPipelineControlFlags[pipelineIndex]);

    results.voxel = &(triLinYStage.results.voxel[pipelineIndex]);
    results.weightsXYZ = &(triLinYStage.results.weightsXYZ[pipelineIndex]);
    results.perPipelineControlFlags
45    = &(triLinYStage.results.perPipelineControlFlags[pipelineIndex]);
    results.perChipControlFlags
        = &(triLinYStage.results.perChipControlFlags);

    cube4 = triLinYStage(cube4);
50
}

```

```

332
} // LocalSetup

5   void TriLinYPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in TriLinYStage
    //////////////////////////////////////////////////////////////////
10   // resize fifo's according to dataset size

    // step delay for a y-step within a block
    beamDelay = (cube4->datasetSizeXYZ.X())
        ) /
15   (numOfChips*numOfPipelinesPerChip);

    // step delay for a y-step between blocks
    blockBeamDelay = beamDelay * blockSize;

    beamVoxelFiFo.SetSize(beamDelay);
    beamWeightsFiFo.SetSize(beamDelay);
20   beamPerPipelineControlFlagsFiFo.SetSize(beamDelay);
    beamPerChipControlFlagsFiFo.SetSize(beamDelay);

    blockBeamVoxelFiFo.SetSize(blockBeamDelay);
    blockBeamWeightsFiFo.SetSize(blockBeamDelay);
25   blockBeamPerPipelineControlFlagsFiFo.SetSize(blockBeamDelay);
    blockBeamPerChipControlFlagsFiFo.SetSize(blockBeamDelay);

    beamVoxelFiFo.Preset(*(inputs.voxel));
    beamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    beamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFlags));
30   beamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    blockBeamVoxelFiFo.Preset(*(inputs.voxel));
    blockBeamWeightsFiFo.Preset(*(inputs.weightsXYZ));
    blockBeamPerPipelineControlFlagsFiFo.Preset(*(inputs.perPipelineControlFla
gs));
35   blockBeamPerChipControlFlagsFiFo.Preset(*(inputs.perChipControlFlags));

    readBigFiFoCounter = readSmallFiFoCounter =
        writeBigFiFoCounter = writeSmallFiFoCounter = -1;
} // PerFrameSetup

40   //////////////////////////////////////////////////////////////////
45   // local computation functions

void TriLinYPipeline::RunForOneClockCycle()
{
    //////////////////////////////////////////////////////////////////
    // buffer reading

    // reset counters
50
55

```

```

333
if (inputs.perChipControlFlags->volumeStart ||
    ((readBigFiFoCounter == 0) &&
     (readSmallFiFoCounter == 0) &&
     (writeBigFiFoCounter == 0) &&
     (writeSmallFiFoCounter == 0))) {
5
    readBigFiFoCounter = beamDelay;
    readSmallFiFoCounter = (blockSize-1) * readBigFiFoCounter;
10
    writeBigFiFoCounter = readBigFiFoCounter;
    writeSmallFiFoCounter = readSmallFiFoCounter;
}

15
///////////////////////////////
// first read from Fifos into results register

// at start of block read from big Fifo
if (readBigFiFoCounter > 0) {
    blockBeamVoxelFiFo.Read(outFiFo);
    blockBeamWeightsFiFo.Read(*(results.weightsXYZ));
20
    blockBeamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlag
s));

    blockBeamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
25
    --readBigFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0)
cout<<"R"<<*(results.voxel)<<volumeSliceGradientFiFo<<endl;
}

30
// in middle and at end of block read from small Fifo
else if (readSmallFiFoCounter > 0) {
    beamVoxelFiFo.Read(outFiFo);
    beamWeightsFiFo.Read(*(results.weightsXYZ));

    beamPerPipelineControlFlagsFiFo.Read(*(results.perPipelineControlFlags));
    beamPerChipControlFlagsFiFo.Read(*(results.perChipControlFlags));
35
    --readSmallFiFoCounter;
    // if (chipIndex == 0 && pipelineIndex == 0)      cout<<"r";
}

40
/////////////////////////////
// computation

// Linear interpolation
// c = w(b-a)+a
//
// w=0 -> c=a
// w=1 -> c=b
45

if (cube4->cubeMode == Cube4Light) {
    // just pick current voxel (as if w=0)

50

```

```

      334
    results.voxel->raw16bit = outFiFo.raw16bit; // w=0;
    // use weights later on in composing
  }

  5   else if (cube4->cubeMode == Cube4Classic) {
    // do real interpolation
    a((int)outFiFo.raw16bit);
    b((int)inputs.voxel->raw16bit);

  10  // use results-weight (they are generated relative to (0,0,0) pos
    results.voxel->raw16bit = (short) ((results.weightsXYZ->Y())*(b-a) +
    a);

    // assign control bits to weights to flow down the rest of the
  15  // pipeline as weights
    results.weightsXYZ->SetY(-1 *
StepDirection[results.perChipControlFlags->yStep]);
  }

  20  /////////////////////////////////
  // buffer writing

  /////////////////////////////////
  // now write to Fifos from inputs register

  25  // at start and in middle of block write to small Fifo
  if (writeSmallFiFoCounter > 0) {
    beamVoxelFiFo.Write(*(inputs.voxel));
    beamWeightsFiFo.Write(*(inputs.weightsXYZ) );

    beamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
  30  beamPerChipControlFlagsFiFo.Write(
    *(inputs.perChipControlFlags));
    --writeSmallFiFoCounter;
    // if (chipIndex == 0 & pipelineIndex == 0) cout<<"w";
  }

  35  // at end of block write to big Fifo of next chip
  else if (writeBigFiFoCounter > 0) {
    ModInt c(chipIndex+1, numOChips);
    int p(pipelineIndex);
    TriLinYPipeline *next(&cube4->triLinY[c].triLinYPipeline[p]);
  40  next->blockBeamVoxelFiFo.Write(*(inputs.voxel));
    next->blockBeamWeightsFiFo.Write(*(inputs.weightsXYZ) );
    next-
    >blockBeamPerPipelineControlFlagsFiFo.Write(*(inputs.perPipelineControlFlags));
    next-
    >blockBeamPerChipControlFlagsFiFo.Write(*(inputs.perChipControlFlags));
  45  --writeBigFiFoCounter;
    //if (chipIndex == 0 & pipelineIndex == 0) cout<<"W";
  }
}

  } // RunForOneClockCycle

```

50

55

```

5   // internal utility functions

10  // end of TriLinYPipeline.C
11  ::::::::::::::
12  cube4/TriLinYPipeline.h
13  ::::::::::::::
14  // TriLinYPipeline.h
15  // (c) Ingmar Bitter '97

16  // Copyright, Mitsubishi Electric Information Technology Center
17  // America, Inc., 1997. All rights reserved.

18  #ifndef _TriLinYPipeline_h_    // prevent multiple includes
19  #define _TriLinYPipeline_h_

20  #include "Misc.h"
21  #include "Object.h"
22  #include "FiFo.h"
23  #include "Voxel.h"
24  #include "Coxel.h"
25  #include "Control.h"
26  #include "FixPointNumber.h"

27  typedef Vector3D<FixPointNumber> FixPointVector3D;

28  class TriLinYStage;
29  class Cube4;

30  class TriLinYPipelineInputs {
31  public: // pointers
32      Voxel *voxel;
33      Vector3D<FixPointNumber> *weightsXYZ;
34      PerChipControlFlags      *perChipControlFlags;
35      PerPipelineControlFlags *perPipelineControlFlags;
36  };

37  class TriLinYPipelineResults {
38  public: // pointers
39      Voxel *voxel;
40      Vector3D<FixPointNumber> *weightsXYZ;
41      PerChipControlFlags      *perChipControlFlags;
42      PerPipelineControlFlags *perPipelineControlFlags;
43  };

44  class TriLinYPipeline : virtual public Object {
45  public:

46      static void      Demo ();

```

```

// constructors & destructors
5   TriLinYPipeline ();
-TriLinYPipeline ();

// show/set data & data properties
// - class Object requirements
10  virtual ostream & Ostream (ostream & ) const;

// - local show/set functions
static void GlobalSetup (const int setNumOfChips,
15  const int setNumOfPipelinesPerChip,
const int setBlockSize);

virtual void LocalSetup (const int setChipIndex,
20  const int setPipelineIndex,
TriLinYStage & triLinYStage);
virtual void PerFrameSetup();
// local computation functions
25  virtual void RunForOneClockCycle();

public:
    TriLinYPipelineInputs inputs;
    TriLinYPipelineResults results;

30  protected:
    Fifo<Voxel> beamVoxelFifo;
    Fifo<FixPointVector3D> beamWeightsFifo;
    Fifo<PerPipelineControlFlags> beamPerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags> beamPerChipControlFlagsFifo;

35  Fifo<Voxel> blockBeamVoxelFifo;
    Fifo<FixPointVector3D> blockBeamWeightsFifo;
    Fifo<PerPipelineControlFlags> blockBeamPerPipelineControlFlagsFifo;
    Fifo<PerChipControlFlags> blockBeamPerChipControlFlagsFifo;

40  Voxel outFifo;
    FixPointNumber a,b;

    int beamDelay, blockBeamDelay;

45  int readSmallFifoCounter, readBigFifoCounter;
    int writeSmallFifoCounter, writeBigFifoCounter;

    static int numOfChips, numOfPipelinesPerChip, blockSize;
    static int maxDataSetSizeX;
50  static Cube4 *cube4;
    int chipIndex, pipelineIndex;

```

```

            friend class Cube4;
};

5   #include "TriLinYStage.h"
# include "Cube4.h"

#ifndef      // _TriLinYPipeline_h_
:::::::::::
cube4/TriLinYStage.C
:::::::::::
// TriLinYStage.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
15  // America, Inc., 1997. All rights reserved.

#include "TriLinYStage.h"

void TriLinYStage::Demo()
{
20    TriLinYStage triLinY;
    cout << endl << "Demo of class " << typeid(triLinY).name();
    cout << endl << "size : " << sizeof(TriLinYStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "TriLinYStage triLinY; = " << triLinY;
    cout << endl << "End of demo of class " << typeid(triLinY).name() << endl;
} // Demo

///////////////////////////////
// constructors & destructors

30  // static first init
int TriLinYStage::numOfChips           = 0;
int TriLinYStage::numOfPipelinesPerChip = 0;
Cube4 *TriLinYStage::cube4 = 0;

35  TriLinYStage::TriLinYStage()
{
    triLinYPipeline = new TriLinYPipeline [numOfPipelinesPerChip];
    results.voxel = new Voxel [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
[numOfPipelinesPerChip];
} // defaultconstructor

40

TriLinYStage::~TriLinYStage()
{
    if (triLinYPipeline) { delete triLinYPipeline; triLinYPipeline=0; }
    if (results.voxel)   { delete results.voxel;   results.voxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0;
}
}

45

```

50

55

```

338
if (results.perPipelineControlFlags) {
    delete results.perPipelineControlFlags;
    results.perPipelineControlFlags=0;
}
} // destructor

///////////////////////////////
10 // show/set data & data properties

ostream & TriLinYStage::Ostream(ostream & os) const
{
    // append TriLinYStage info to os
15    os << typeid(*this).name() << "@" << (void *) this;
        os << endl << "numOfChips" = " << numOfChips;
        os << endl << "numOfPipelinesPerChip" = " << numOfPipelinesPerChip;
        os << endl << "chipIndex" = " << chipIndex;

20    // return complete os
    return os;

} // Ostream

25 /////////////////////
// show/set data & data properties
//
// - local show/set functions

30 void TriLinYStage::GlobalSetup(const int setNumOfChips,
                                const int setNumOfPipelinesPerChip,
                                const int setBlockSize,
35                                Cube4 *setCube4)
{
    numOfChips = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
40    TriLinYPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip,
                                setBlockSize);
    cube4 = setCube4;
} // GlobalSetup

45 void TriLinYStage::LocalSetup(const int setChipIndex)
{
    chipIndex = setChipIndex;
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
50        triLinYPipeline[p].LocalSetup(chipIndex,p,*this);
}

```

55

339

```

        }
    } // LocalSetup

5

void TriLinYStage::PerFrameSetup()
{
    int p;
    // reset pipeline registers
10   for (p=0; p<numOfPipelinesPerChip; ++p) {
        results.voxel[p].raw16bit = 0;
        results.weightsXYZ[p](0,0,0);
        results.perPipelineControlFlags[p].Reset();
    }
    results.perChipControlFlags.Reset();

15   for (p=0; p<numOfPipelinesPerChip; ++p) {
        triLinYPipeline[p].PerFrameSetup();
    }

20   // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

25   /////////////////////////////////
// local computation functions

void TriLinYStage::RunForOneClockCycle()
{
    // communication

30    // computation
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        triLinYPipeline[p].RunForOneClockCycle();
    }
} // RunForOneClockCycle

35   /////////////////////////////////
// internal utility functions

40   // end of TriLinYStage.C
:::::::::::
cube4/TriLinYStage.h
:::::::::::
// TriLinYStage.h
// (c) Ingmar Bitter '97

45   // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

    #ifndef _TriLinYStage_h_      // prevent multiple includes
50

```

55

```

#define _TriLinYStage_h_

5   #include "Misc.h"
    #include "Object.h"
    #include "Vector3D.h"
    #include "Voxel.h"
    #include "Control.h"
10  #include "TriLinYPipeline.h"
    #include "FixPointNumber.h"
    #include "Cube4.h"

    class Cube4;

15  class TriLinYStageInputs {
public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      *perChipControlFlags;
20  PerPipelineControlFlags *perPipelineControlFlags;
};

25  class TriLinYStageResults {
public: // arrays
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags      perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

30

35  class TriLinYStage : virtual public Object {
public:

    static void      Demo ();

        // constructors & destructors
        TriLinYStage ();
        ~TriLinYStage ();

40  // show/set data & data properties
        // - class Object requirements
        virtual ostream & Ostream (ostream &) const;

        // - local show/set functions
45  static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           const int setBlockSize,
                           Cube4 *setCube4);
                           virtual void LocalSetup (const int setChipIndex);

50
55

```

```

341
    virtual void PerFrameSetup ();
    // local computation functions
5     virtual void RunForOneClockCycle();

public:
    TriLinYPipeline *triLinYPipeline;
    TriLinYStageInputs inputs;
    TriLinYStageResults results;

10    protected:
        static int    numOfChips, numOfPipelinesPerChip;
        static Cube4 *cube4;
        int          chipIndex; // only for debugging purpose

15    friend class TriLinYPipeline;
};

#endif // _TriLinYStage_h_
::::::::::
20    cube4/TriLinZPipeline.C
::::::::::
// TriLinZPipeline.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
25 // America, Inc., 1997, All rights reserved.

#include "TriLinZPipeline.h"

void TriLinZPipeline::Demo()
{
30    TriLinZPipeline triLinZ;
    cout << endl << "Demo of class " << typeid(triLinZ).name();
    cout << endl << "size : " << sizeof(TriLinZPipeline) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "TriLinZPipeline triLinZ; = " << triLinZ;
    cout << endl << "End of demo of class " << typeid(triLinZ).name() << endl;
35 } // Demo

///////////////////////////////
// constructors & destructors
40
// static first init
int TriLinZPipeline::numOfChips           = 0;
int TriLinZPipeline::numOfPipelinesPerChip = 0;
Cube4 *TriLinZPipeline::cube4             = 0;

45 TriLinZPipeline::TriLinZPipeline()
{
} // constructor

50

```

```

342
TriLinZPipeline::~TriLinZPipeline()
{
5   } // destructor

///////////////////////////////
// show/set data & data properties

10
ostream & TriLinZPipeline::Ostream(ostream & os) const
{
    // append TriLinZPipeline info to os
    os << typeid(*this).name() << "@" << (void *) this;
15    os << endl << "  numOfChips      = " << numOfChips;
    os << endl << "  numOfPipelinesPerChip = " << numOfPipelinesPerChip;
    os << endl << "  chipIndex        = " << chipIndex;

    // return complete os
20    return os;
} // Ostream

///////////////////////////////
25 // show/set data & data properties
//
// - local show/set functions

30 void TriLinZPipeline::GlobalSetup(const int setNumOfChips,
                                    const int setNumOfPipelinesPerChip)
{
    numOfChips      = setNumOfChips;
    numOfPipelinesPerChip = setNumOfPipelinesPerChip;
35 } // GlobalSetup

40 void TriLinZPipeline::LocalSetup(const int setChipIndex,
                                    const int setPipelineIndex,
                                    TriLinZStage & triLinZStage)
{
    chipIndex = setChipIndex;
    pipelineIndex = setPipelineIndex;
45    inputs.voxel0 = &(triLinZStage.inputs.voxel0[pipelineIndex]);
    inputs.voxel1 = &(triLinZStage.inputs.voxel1[pipelineIndex]);
    inputs.weightsXYZ = &(triLinZStage.inputs.weightsXYZ[pipelineIndex]);
    inputs.perChipControlFlags
50    = triLinZStage.inputs.perChipControlFlags;
    inputs.perPipelineControlFlags

```

55

```

343
=
&(triLinZStage.inputs.perPipelineControlFlags[pipelineIndex]);
5    results.voxel = &(triLinZStage.results.voxel[pipelineIndex]);
    results.perPipelineControlFlags
        = &(triLinZStage.results.perPipelineControlFlags[pipelineIndex]);
    cube4 = triLinZStage(cube4);
10 } // LocalSetup

void TriLinZPipeline::PerFrameSetup()
{
    // reset pipeline registers already done in TriLinZStage
15 } // PerFrameSetup

///////////////////////////////
// local computation functions
20 void TriLinZPipeline::RunForOneClockCycle()
{
    /*
        / (z)
        /
25    /voxB (seen late (recently) during processing => take from FiFo0 )
        /
        /voxA (seen early (long ago) during processing => take from FiFol )
        /
        +-----> (x)
        |
        |
        V(y)
    */
35
    // Linear interpolation
    // c = w(b-a)+a
    //
    // w=0 -> c=a
    // w=1 -> c=b
40

    if (cube4->cubeMode == Cube4Light) {
        // just pick current voxel (as if w=0)
        results.voxel->raw16bit = inputs.voxell->raw16bit; // w=0;
45        // use weights later on in composing
    }

    else if (cube4->cubeMode == Cube4Classic) {
        // do real interpolation
        a((int)inputs.voxell->raw16bit);
50
}

```

```

344
      b((int)delayVoxel0.raw16bit);

5       results.voxel->raw16bit = (short)(inputs.weightsXYZ->z() * (b - a) +
a);
      }

10      // complete time-lineup between slice buffers
// by delaying the data from before the FiFo by one more cycle

11      delayVoxel0 = *(inputs.voxel0);

12  } // RunForOneClockCycle

13
14  /////////////////////////////////
15  /////////////////////////////////
16  // internal utility functions

17
18  // end of TriLinZPipeline.C
19  ::::::::::::::::::::
20  cube4/TriLinZPipeline.h
21  ::::::::::::::::::::
22  // TriLinZPipeline.h
23  // (c) Ingmar Bitter '97

24
25  // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

26  #ifndef _TriLinZPipeline_h_    // prevent multiple includes
27  #define _TriLinZPipeline_h_

28
29  #include "Misc.h"
30  #include "Object.h"
31  #include "Voxel.h"
32  #include "Coxel.h"
33  #include "Control.h"
34  #include "FixPointNumber.h"

35
36  class TriLinZStage;
37  class Cube4;

38
39  class TriLinZPipelineInputs {
40  public: // pointers
41      Voxel *voxel0;
42      Voxel *voxel1;
43      Vector3D<FixPointNumber> *weightsXYZ;
44      PerChipControlFlags *perChipControlFlags;
45      PerPipelineControlFlags *perPipelineControlFlags;
};

46

47  class TriLinZPipelineResults {

48
49
50
51
52
53
54
55

```

345

```

public: // pointers
    Voxel *voxel;
    Vector3D<FixPointNumber> *weightsXYZ;
    PerChipControlFlags *perChipControlFlags;
    PerPipelineControlFlags *perPipelineControlFlags;
};

10   class TriLinZPipeline : virtual public Object {
public:

    static void Demo ();

15   // constructors & destructors
    TriLinZPipeline ();
    ~TriLinZPipeline ();

    // show/set data & data properties
    // - class Object requirements
20   virtual ostream & Ostream (ostream &) const;

    // - local show/set functions
    static void GlobalSetup (const int setNumOfChips,
25   const int setNumOfPipelinesPerChip);

    virtual void LocalSetup(const int setChipIndex,
                           const int setPipelineIndex,
30   TriLinZStage & triLinZStage);
    virtual void PerFrameSetup();
    // local computation functions
    virtual void RunForOneClockCycle();

35   public:
        TriLinZPipelineInputs inputs;
        TriLinZPipelineResults results;

    protected:
40   Voxel delayVoxel0; // delay voxel from SliceVoxelFiFo0 for one extra
cycle
        FixPointNumber a,b;
        static int numOfChips, numOfPipelinesPerChip;
        static Cube4 *cube4;
        int chipIndex, pipelineIndex;
45   friend class Cube4;
};

#include "TriLinZStage.h"
#include "Cube4.h"

```

55

346

```

#endif      // _TriLinZPipeline_h_
:::::::::::
cube4/TriLinZStage.C
:::::::::::
// TriLinZStage.C
// (c) Ingmar Bitter '97
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

#include "TriLinZStage.h"

void TriLinZStage::Demo()
{
    TriLinZStage triLinZ;
    cout << endl << "Demo of class " << typeid(triLinZ).name();
    cout << endl << "size : " << sizeof(TriLinZStage) << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "TriLinZStage triLinZ; = " << triLinZ;
    cout << endl << "End of demo of class " << typeid(triLinZ).name() << endl;
} // Demo

///////////////////////////////
// constructors & destructors

// static first init
int TriLinZStage::numOfChips          = 0;
int TriLinZStage::numOfPipelinesPerChip = 0;
Cube4 *TriLinZStage::cube4 = 0;

TriLinZStage::TriLinZStage()
{
    triLinZPipeline = new TriLinZPipeline [numOfPipelinesPerChip];
    results.voxel = new Voxel [numOfPipelinesPerChip];
    results.weightsXYZ = new Vector3D<FixPointNumber> [numOfPipelinesPerChip];
    results.perPipelineControlFlags = new PerPipelineControlFlags
        [numOfPipelinesPerChip];
} // defaultconstructor

TriLinZStage::~TriLinZStage()
{
    if (triLinZPipeline) { delete triLinZPipeline; triLinZPipeline=0; }
    if (results.voxel) { delete results.voxel; results.voxel=0; }
    if (results.weightsXYZ) { delete results.weightsXYZ; results.weightsXYZ=0; }

    if (results.perPipelineControlFlags) {
        delete results.perPipelineControlFlags;
        results.perPipelineControlFlags=0;
    }
} // destructor

```

50

55

```

5   // show/set data & data properties

10  ostream & TriLinZStage::Ostream(ostream & os) const
11  {
12      // append TriLinZStage info to os
13      os << typeid(*this).name() << "@" << (void *) this;
14      os << endl << " numOfChips           = " << numOfChips;
15      os << endl << " numOfPipelinesPerChip = " << numOfPipelinesPerChip;
16      os << endl << " chipIndex            = " << chipIndex;

17  // return complete os
18  return os;
19 } // Ostream

20 // show/set data & data properties
21 //
22 // - local show/set functions

23 void TriLinZStage::GlobalSetup(const int setNumOfChips,
24
25     const int setNumOfPipelinesPerChip,
26
27     Cube4 *setCube4)
28 {
29     numOfChips          = setNumOfChips;
30     numOfPipelinesPerChip = setNumOfPipelinesPerChip;
31     TriLinZPipeline::GlobalSetup(numOfChips, numOfPipelinesPerChip);
32     cube4 = setCube4;
33 } // GlobalSetup

34 void TriLinZStage::LocalSetup(const int setChipIndex)
35 {
36     chipIndex = setChipIndex;
37     for (int p=0; p<numOfPipelinesPerChip; ++p) {
38         triLinZPipeline[p].LocalSetup(chipIndex,p,*this);
39     }
40 } // LocalSetup

41 void TriLinZStage::PerFrameSetup()
42 {
43     int p;
44     // reset pipeline registers
45     for (p=0; p<numOfPipelinesPerChip; ++p) {
46         results.voxel[p].raw16bit = 0;
47     }
48 }
```

```

348         results.weightsXYZ[p](0,0,0);
      results.perPipelineControlFlags[p].Reset();
5   }
results.perChipControlFlags.Reset();
    for (p=0; p<numOfPipelinesPerChip; ++p) {
        triLinZPipeline[p].PerFrameSetup();
    }
10
    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }
} // PerFrameSetup

15
////////////////////////////// local computation functions
void TriLinZStage::RunForOneClockCycle()
{
    // computation
20
    for (int p=0; p<numOfPipelinesPerChip; ++p) {
        results.weightsXYZ[p] = inputs.weightsXYZ[p];
        results.perPipelineControlFlags[p] =
inputs.perPipelineControlFlags[p];
        triLinZPipeline[p].RunForOneClockCycle();
    }
    results.perChipControlFlags = *(inputs.perChipControlFlags);
} // RunForOneClockCycle

30
////////////////////////////// internal utility functions

    // end of TriLinZStage.C
    ::::::::::::::
35    cube4/Voxel.C
    ::::::::::::::
    // Voxel.C
    // (c) Ingmar Bitter '97
    // Copyright, Mitsubishi Electric Information Technology Center
40    // America, Inc., 1997. All rights reserved.

    #include "Voxel.h"
    ostream & operator << (ostream & os, const Voxel & v)
45
    {
        short s(v.raw16bit);
        os << "[" << s << "|" << s%32 << "," << s/32%32 << "," << s/32/32%32 <<
    "] ";
}

```

50

55

349

```

        return os;
    } // operator <<

5

// end of Voxel.C
::::::::::
cube4/TriLinZStage.h
::::::::::
10 // TriLinZStage.h
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.
15

#ifndef _TriLinZStage_h_      // prevent multiple includes
#define _TriLinZStage_h_

20 #include "Misc.h"
#include "Object.h"
#include "Voxel.h"
#include "Control.h"
#include "TriLinZPipeline.h"
#include "FixPointNumber.h"
25 #include "Cube4.h"

    class Cube4;

    class TriLinZStageInputs {
public: // pointers
30     Voxel *voxel0; // from SliceVoxelFiFo0
     Voxel *voxel1; // from SliceVoxelFiFo1
     Vector3D<FixPointNumber> *weightsXYZ;
     PerChipControlFlags    *perChipControlFlags;
     PerPipelineControlFlags *perPipelineControlFlags;
35     };

    class TriLinZStageResults {
public: // arrays
40     Voxel *voxel;
     Vector3D<FixPointNumber> *weightsXYZ;
     PerChipControlFlags    perChipControlFlags;
     PerPipelineControlFlags *perPipelineControlFlags;
    };
45

    class TriLinZStage : virtual public Object {
public:

50     static void      Demo ();

         // constructors & destructors
         TriLinZStage ();

```

55

```

350
~TriLinZStage ();

5   // show/set data & data properties
     // - class Object requirements
     virtual ostream & Ostream (ostream & ) const;

10  // - local show/set functions
     static void GlobalSetup (const int setNumOfChips,
                           const int setNumOfPipelinesPerChip,
                           Cube4 *setCube4);
     virtual void LocalSetup (const int setChipIndex);
15  virtual void PerFrameSetup ();
     // local computation functions
     virtual void RunForOneClockCycle();

20  public:
     TriLinZPipeline *triLinZPipeline;
     TriLinZStageInputs inputs;
     TriLinZStageResults results;

25  protected:
     static int    numOfChips, numOfPipelinesPerChip;
     static Cube4 *cube4;
     int          chipIndex; // only for debugging purpose

30  friend class TriLinZPipeline;
};

35  #endif // _TriLinZStage_h_
:::::::
cube4/VoxMem.C
:::::::
// VoxMem.C
// (c) Ingmar Bitter '97

40  // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45  // DRAM Voxel Memory of one pipeline
// in : 1/numOfPipeline fraction of the dataset
// out: random access to any of these voxels

50  #include "VoxMem.h"
void VoxMem::Demo()
{
    VoxMem mem;
    cout << endl << "Demo of class " << typeid(mem).name();
    cout << endl << "size : " << sizeof(VoxMem) << " Bytes";
    cout << endl << "public member functions:";
    mem(3)=65; cout << endl << " mem(3)=65      = " << mem(3);
}

```

55

```

      cout << endl << "VoxMem::Demo()      end" << endl;
} // Demo
5

///////////////////////////////
// constructors & destructors

10 VoxMem::VoxMem()
    : size(0), mem(0)
{
} // constructor

15 VoxMem::~VoxMem()
{
    if (mem) { delete mem; mem=0; }
} // destructor

20 /////////////////////
// show/set data & data properties

25 ostream & VoxMem::Ostream(ostream & os) const
{
    // append Control info to os
    os << typeid(*this).name() << "@" << (void *) this;
    os << endl << " size = " << size;

    // return complete os
    return os;
}

30 } // Ostream

35 // - local show/set functions

35 bool VoxMem::Init (unsigned int setSize)
{
    size = setSize;
    if (mem) delete mem;
    mem = new Voxel [size];
    if (!mem) ERROR("OutOfMemoryError");

    // print debug info
    //static bool first(true); if (first) { cout<<this<<endl; first=false; }

    return true;
40 } // Init

45 Voxel & VoxMem::operator () (unsigned int index)
{
50

```

352

```

// subscript a array value
// check that index is valid
5   if (index >= size) {
        cout << index << "index:size" << size;
        ERROR("TooLargeArrayIndexError!");
    }

10    // return reference to requested element
        return mem[index];
} // operator ()


15    // end of VoxMem.C
::::::
cube4/VoxMem.h
::::::
// VoxMem.h
// (c) Ingmar Bitter '97
20
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// DRAM Voxel Memory of one pipeline
25
// in : 1/numOfChips fraction of the dataset
// out: bustmode access to blocks of voxels

#ifndef _VoxMem_h_           // prevent multiple includes
#define _VoxMem_h_

30
#include "Object.h"
#include "Misc.h"
#include "Voxel.h"

35
class VoxMem : virtual public Object {
public:

    static void      Demo ();

    // constructors & destructors
40
    VoxMem();
    virtual ~VoxMem();

    // show/set data & data properties
    // - class Object requirements
    virtual       ostream &   Ostream (ostream &) const;

45
    virtual       bool      Init      (unsigned int setSize);
    virtual /*inline*/ Voxel & operator () (unsigned int index);

    // local computation functions

50
protected:
    Voxel *mem;

```

55

353

```

    unsigned int size;
};

5      #endif      // _VoxMem_h_
:::::::
cube4/Voxel.h
:::::::
// Voxel.h
10     // (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15     #ifndef _Voxel_h_ // prevent multiple includes
#define _Voxel_h_

#include <iostream.h> // cout
#include "Global.h" // cout

20     union Voxel {
    unsigned char raw8bit;
    unsigned short raw16bit;
    struct test {
        unsigned x      : 5;
        unsigned y      : 5;
        unsigned z      : 5;
    };
    struct Format0 {
        unsigned data    : 8;
    };
    struct Format1 {
        unsigned data    : 16;
    };
    struct Density12_Index4 {
        unsigned data    : 12;
        unsigned segmentIndex : 4;
    };
    //    Voxel (unsigned char c) { raw8bit = c; };
    //    Voxel (unsigned short s) { raw16bit = s; };
    Voxel (unsigned short i=0) { raw16bit = i; };
    friend ostream & operator << (ostream & os, const Voxel & v);
}; // union Voxel

45     #endif      // _Voxel_h_
:::::::
cube4/doc/CoordinateSystems.doc
:::::::
Frist of all, from now on we would like to use the following conventions
for the coordinate systems we discuss in Cube-4, to make the discussions
50     easier and the explanations clearer.
```

Abbreviations:

```
=====

```

```
5      n : # samples along one axis of the dataset
      p : # parallel pipelines

      Ns : Slice number
      NB : Beam number
      Nb : Partial beam number
10
      (i,k)           : Memory Space coordinates
      (u,v,w) = (Ud,Vd,Wd) : Dataset   Space coordinates (righthanded)
      (x,y,z) = (Xp,Yp,Zp) : Pipeline   Space coordinates (righthanded)
      (u,v,w) = (Ub,Vb,Wb) : Baseplane Space coordinates (righthanded)
15      (x,y,z) = (Xi,Yi,Zi) : Image     Space coordinates (righthanded)
```

Memory Space:

```
=====

```

```
20      i ^                                Conversions from dataset coordinates (Ud,Vd,Wd):
      |    +-+    +-+    +-+
      a | <| |> <| |> <| |>    full parallel pipeline (p == n):
      d | <| R |> <| R |> <| R |>    i = v + w*n
      d | <| A |> <| A |> <| A |>    k = (u+v+w) % n
      r | <| M |> <| M |> <| M |>
25      e | <| |> <| |> <| |>    partial beam partitioned pipeline (p < n):
      s | +-+    +-+    +-+    i = Nb + NB * (n/p) + Ns * (n^2/p)
      s +----->    k = (u+v+w) % p
      memory bank number k
```

```
30      eg: 4x4x4 dataset stored in 4 memory modules (full parallel)
          the entries below represent [wvu] dataset coordinates
```

```
i ^
35      15| 332 333 330 331
      14| 323 320 321 322
      13| 310 311 312 313
      12| 301 302 303 300
      |
      11| 233 230 231 232
      10| 220 221 222 223
40      9 | 211 212 213 210
      8 | 202 203 200 201
      |
      7 | 130 131 132 133
      a 6 | 121 122 123 120
      d 5 | 112 113 110 111
45      d 4 | 103 100 101 102
      r |
      e 3 | 031 032 033 030
      s 2 | 022 023 020 021
      s 1 | 013 010 011 012
50      0 | 000 001 002 003
      +----->
```

355

0 1 2 3
memory bank number k

5

10

Dataset Space:

— — — — —

```

15      (u,v,w) == (Ud,Vd,Vd)           Conversion from pipeline coordinates
                                         to Dataset coordinates
                                         |
                                         w/ |          (Ud)          (Xp)
                                         /           (Vd) = Mpd * (Yp)
                                         +---+---+---+---+          (Wd)          (Zp)
                                         +---+---+---+---+|          (1 )          (1 )
                                         +---+---+---+---+| + u
                                         +---+---+---+---+---+>
                                         |   |   |   |   | + | +
                                         +---+---+---+---+| + | 25
                                         |   |   |   |   | + | +
                                         +---+---+---+---+| + | +
                                         |   |   |   |   | + | +
                                         +---+---+---+---+| + | +
                                         |   |   |   |   | + | +      Mpd = 4x4 matrix
                                         +---+---+---+---+| + | 30
                                         |   |   |   |   | + | +      Mpd = Mdp^(-1)
                                         +---+---+---+---+
                                         |
                                         v
                                         V

```

35 Origin always at upper left corner of storage order frontmost slice.
Unit length == distance between samples along one axies.

Pipeline Space:

=====
=====

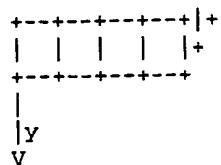
```

40      (x,y,z) == (Xp,Yp,Zp)           Conversion from dataset coordinates
                                         to pipeline coordinates
                                         (Xp)          (Ud)
                                         (Yp) = Mdp * (Vd)
                                         (Zp)          (Wd)
                                         (1 )         (1 )
45
                                         (1 )         (1 )
                                         x
                                         +----->   Conversion from baseplane coordinates
                                         | | | | | +|+|   to pipeline coordinates
                                         +-----+ +|+|   (Xp)          (Ub)
                                         | | | | | +|+|   (Yp) = Mbp * (Vb)
                                         +-----+ +|+|   (Zp)          (Wb)
                                         | | | | | +|+|   (1 )         (1 )
50
                                         (1 )

```

55

356



Mpd, Mpb = 4x4 matrices
 $Mpd = Mdp^{-1}$
 $Mpb = Mbp^{-1}$

5

10 Origin always at upper left corner of view direction frontmost slice.
 Unit length == distance between samples along one axies.

15

20

Baseplane Space:

=====

(u, v, w) == (Ub, Vb, Vb)

Conversion from pipeline coordinates
to baseplane coordinates

(Ub)	(Xp)
(Vb)	= Mpb * (Yp)
(Wb)	(Zp)
(1)	(1)

30

Conversion from image coordinates
to baseplane coordinates

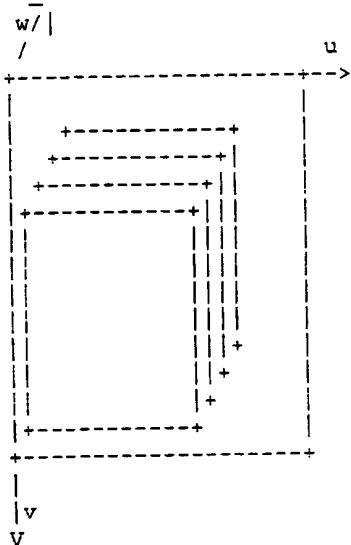
(Ub)	(Xi)
(Vb)	= Mib * (Yi)
(Wb)	(Zi)
(1)	(1)

35

Mpb, Mib = 4x4 matrices

 $Mpb = Mbp^{-1}$
 $Mib = Mbi^{-1}$

40



45

Unit length == distance between samples along one axies.

50

Image Space:

=====

55

Unit length == distance between image pixels along one axies.

```
30
::::::::::
cube4/doc/HOWTO-addNewPipeStage.doc
::::::::::
HOWTO-addNewPipelineStage
=====
35
Ingmar Bitter 4-23-97

New Pipeline stage: Foo

40
o Take an existing Pipeline stage Bar with similar' in/outputs

o copy BarStage.C to FooStage.C
o copy BarStage.h to FooStage.h
o copy BarPipeline.C to FooPipeline.C
o copy BarPipeline.h to FooPipeline.h

45
o search and replace all bar's by foo's and Bar's by Foo's in the new
files

FooStage.h:
o update class FooStageInputs // pointers to previous results
50
```

358

```

o update class FooStageResults //           (dynamic) arrays of results
o put(delete) (un)necessarry class variables in protected section
5

FooPipeline.h:
o update class FooStageInputs // pointers to Stage array
o update class FooStageResults // pointers to Stage array
o put(delete) (un) necessarry class variables in protected section

10
FooStage.C:
o check for completeness: // static first init
o constructor/destructor: make updates according to updates of
inputs/results class, keep array of pipelines !
o check in SetupFunctions, that all local variables + in/output are
15 initialized
o RunForOneClockCycle(): do the copy stuff and communication stuff
here, real computation is done in the pipelines

20
FooPipeline.C:
o check for completeness: // static first init
o check in SetupFunctions, that all local variables + in/output are
initialized
o in LocalSetup() connect input/output pointers to appropiate members
of Stage class
o RunForOneClockCycle(): do the real computation of this pipeline

25

Cube4.h:
o #include "Foo*.h"
o protected:
30     ... DebugFoo(...) ...
     Foo *foo;

Cube4Debug.C:
o Add a FooDebug() function

35
Cube4.C:
o constructor
// allocate memory for pipeline objects:
...
40 Foo::GlobalSetup()
...
     foo = new Foo [numOfChips];
...
// setup pipeline connections
...
45 foo[c].inputs. ... = stageBeforeFoo[c].results. ...
     foo[c].LocalSetup(c);
...
     stageAfterFoo[c].inputs. ... = foo[c].results. ...
...
50
o destructor
     if (foo)          { delete foo;         foo      =0; }

55

```

359

```

o PerFrameSetup()
...
5   for (c=0; c<numOfChips; ++c) { foo[c].PerFrameSetup(); }
...

o RunPipeline()
...
10  for (c=0; c<numOfChips; ++c) { foo[c].RunForOneClockCycle(); }
    DebugFoo(clk);
    ...

15  Makefile:
o Add "FooStage.C FooPipeline.C" to the SRC line
o type make depend
o type make all

20  ::::::::::::::::::::
cube4/doc/blocking.doc
::::::::::::::::::
Cube-4 Blocking
=====

25  (A) driver interface implications
-----

The complete volume of at most 256^3 voxels is grouped into 8^3 blocks
which are distributed skewed across the memory modules.

30  Each 8^3 block is subdivided into 64 mini-blocks of size 2^3. All
mini-blocks belonging to a particular block reside in the same memory
module and occupy a contiguous chunk of linear memory. There is no
skewing inside blocks.

35  The one board implementation will have 4 VoxMem DRAMs holding 32 MB, 4
CoxMem DRAMs holding 0.5 MB and 4 Cube-4 chips with 4 rendering
pipelines per chip. Therefore, volume management has to keep the size
of the to be rendered dataset a multiple of 4*8=32 in the horizontal
axis (in pipeline coordinate system). In y direction any multiple of 2
(mini-block granularity) is ok while in z it has to be a multiple of 4
(FinalCoxelBuffer readout delay). However, the horizontal axis in
the pipeline coordinate system could be any of the major axis in the
dataset coordinate system, so the easiest is to keep all dataset
dimensions a multiple of 32.

40  45  Using tiling of partial beams of mini-blocks across the volume allows
horizontally oversized datasets. Using horizontal sectioning allows
vertically oversized datasets. Finally using the slice buffers allows
for really deep volumes. So big long sticks can be computed in only
one pass without change of the algorithm. Unfortunately, some of the
internal buffers grow. The maximum values are:
      256 wide (no extra storage for y- and z-step buffers)
      512 wide (16KB extra storage for y- and z-step buffers)

50

```

55

360

1024 wide (33KB extra storage for y- and z-step buffers)
 16K high (no extra storage on Cube-4 chip, CoxMem grows by 1 MB)
 16K deep (no extra storage on Cube-4 chip, CoxMem grows by 1 MB)
 Thus, y and z oversizes up to 32x32x16K can be handled by the hardware without modifications, x oversizes probably require the driver to split the rendering into 256 wide chunks, which will then be composited in 2D by the driver.

Really huge volumes are best cut up into 256 wide, 32 high (one section) and 2048 deep bricks, combining those bricks has to be done by the driver (using 3D (polygon) graphics board).

Sub volumes -- a zoomed-in region of a big dataset (e.g. tumor in brain), or initially small datasets (e.g. leg of game figure) should be at least 32³ to use the full available bandwidth. If the sub volume is a zoom of a big volume, it can be positioned with the granularity of the 8³ blocks. Finer changes can be achieved by specifying additional axis aligned cutting planes.

6 axis aligned cutting planes + one arbitrary cutting plane can be supported by the hardware.

Dynamically growing volumes generated by a phantom operated clay spray tool keep most of their data constant. Only small regions have to be updated. If the to-be-updated region exceeds the current volume a new slice of 8³ blocks has to be allocated and filled with zeros and data from the new region. If the slice is not perpendicular to the view and the complete object is to be rendered, an increase by 4 slices of 8³ blocks is necessary (keep horizontal volume extend a multiple of 32).

(B) internal buffer size implications (#copies x depth x width)

2x 8x2 bytes mini-block read buffer
 4x256x2 beam of mini-blocks buffer

4x 2x2	TriLinX
4x 2x2	GradX
4x 2x2	GradZLinX
4x 2x4	ComposSelX
4x 2x4	ComposLinX

4x 64x2	TriLinY
4x 64x2	GradZLinY
8x 64x2	GradY
4x 64x4	ComposSelY
4x 64x4	ComposLinY

8x 16x2	2 SliceVoxelFiFo (inside slabs)
8x512x2	2 SliceVoxelFiFo (between slabs)

361

4x 16x2	GradZLinZ (inside slabs)
4x512x2	GradZLinZ (between slabs)
5	4x512x4 ComposBuffer
	3x 64x4 FinalCoxelBuffer
10	total: 26 K bytes
	(C) Urs' requirements for on chip shader storage
15	----- 6x512x3 bytes reflectance tables (RGB intensities) 2x256x1 distortion compensation table (ROM).
	total: 9.5 K bytes
20	:::::::::::::: cube4/linramp.txt :::::::::::::
	0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 25 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 30 10 10 10 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 35 15 15 15 15 15 16 16 16 16 16 17 17 17 17 17 18 18 18 18 18 19 19 19 19 19 40 20 20 20 20 20 21 21 21 21 21 22 22 22 22 22 23 23 23 23 23 24 24 24 24 24 45 25 25 25 25 25 26 26 26 26 26 27 27 27 27 27 28 28 28 28 28 29 29 29 29 29 50 30 30 30 30 30 31 31 31 31 31 32 32 32 32 32 33 33 33 33 33

	34	34	34	34	34
5	35	35	35	35	35
	36	36	36	36	36
	37	37	37	37	37
	38	38	38	38	38
	39	39	39	39	39
	40	40	40	40	40
10	41	41	41	41	41
	42	42	42	42	42
	43	43	43	43	43
	44	44	44	44	44
	45	45	45	45	45
15	46	46	46	46	46
	47	47	47	47	47
	48	48	48	48	48
	49	49	49	49	49
	50	50	50	50	50
	51	51	51	51	51
20	52	52	52	52	52
	53	53	53	53	53
	54	54	54	54	54
	55	55	55	55	55
	56	56	56	56	56
25	57	57	57	57	57
	58	58	58	58	58
	59	59	59	59	59
	60	60	60	60	60
	61	61	61	61	61
30	62	62	62	62	62
	63	63	63	63	63
	64	64	64	64	64
	65	65	65	65	65
	66	66	66	66	66
	67	67	67	67	67
35	68	68	68	68	68
	69	69	69	69	69
	70	70	70	70	70
	71	71	71	71	71
	72	72	72	72	72
40	73	73	73	73	73
	74	74	74	74	74
	75	75	75	75	75
	76	76	76	76	76
	77	77	77	77	77
	78	78	78	78	78
45	79	79	79	79	79
	80	80	80	80	80
	81	81	81	81	81
	82	82	82	82	82
	83	83	83	83	83
50	84	84	84	84	84
	85	85	85	85	85
	86	86	86	86	86

	87	87	87	87	87
	88	88	88	88	88
5	89	89	89	89	89
	90	90	90	90	90
	91	91	91	91	91
	92	92	92	92	92
	93	93	93	93	93
10	94	94	94	94	94
	95	95	95	95	95
	96	96	96	96	96
	97	97	97	97	97
	98	98	98	98	98
	99	99	99	99	99
15	100	100	100	100	100
	101	101	101	101	101
	102	102	102	102	102
	103	103	103	103	103
	104	104	104	104	104
20	105	105	105	105	105
	106	106	106	106	106
	107	107	107	107	107
	108	108	108	108	108
	109	109	109	109	109
25	110	110	110	110	110
	111	111	111	111	111
	112	112	112	112	112
	113	113	113	113	113
	114	114	114	114	114
30	115	115	115	115	115
	116	116	116	116	116
	117	117	117	117	117
	118	118	118	118	118
	119	119	119	119	119
35	120	120	120	120	120
	121	121	121	121	121
	122	122	122	122	122
	123	123	123	123	123
	124	124	124	124	124
	125	125	125	125	125
40	126	126	126	126	126
	127	127	127	127	127
	128	128	128	128	128
	129	129	129	129	129
	130	130	130	130	130
45	131	131	131	131	131
	132	132	132	132	132
	133	133	133	133	133
	134	134	134	134	134
	135	135	135	135	135
50	136	136	136	136	136
	137	137	137	137	137
	138	138	138	138	138
	139	139	139	139	139

	140	140	140	140	140	140
5	141	141	141	141	141	
	142	142	142	142	142	
	143	143	143	143	143	
	144	144	144	144	144	
	145	145	145	145	145	
	146	146	146	146	146	
10	147	147	147	147	147	
	148	148	148	148	148	
	149	149	149	149	149	
	150	150	150	150	150	
	151	151	151	151	151	
15	152	152	152	152	152	
	153	153	153	153	153	
	154	154	154	154	154	
	155	155	155	155	155	
	156	156	156	156	156	
	157	157	157	157	157	
20	158	158	158	158	158	
	159	159	159	159	159	
	160	160	160	160	160	
	161	161	161	161	161	
	162	162	162	162	162	
25	163	163	163	163	163	
	164	164	164	164	164	
	165	165	165	165	165	
	166	166	166	166	166	
	167	167	167	167	167	
30	168	168	168	168	168	
	169	169	169	169	169	
	170	170	170	170	170	
	171	171	171	171	171	
	172	172	172	172	172	
	173	173	173	173	173	
35	174	174	174	174	174	
	175	175	175	175	175	
	176	176	176	176	176	
	177	177	177	177	177	
	178	178	178	178	178	
40	179	179	179	179	179	
	180	180	180	180	180	
	181	181	181	181	181	
	182	182	182	182	182	
	183	183	183	183	183	
45	184	184	184	184	184	
	185	185	185	185	185	
	186	186	186	186	186	
	187	187	187	187	187	
	188	188	188	188	188	
	189	189	189	189	189	
50	190	190	190	190	190	
	191	191	191	191	191	
	192	192	192	192	192	

	193 193 193 193 193
5	194 194 194 194 194
	195 195 195 195 195
	196 196 196 196 196
	197 197 197 197 197
	198 198 198 198 198
	199 199 199 199 199
10	200 200 200 200 200
	201 201 201 201 201
	202 202 202 202 202
	203 203 203 203 203
	204 204 204 204 204
15	205 205 205 205 205
	206 206 206 206 206
	207 207 207 207 207
	208 208 208 208 208
	209 209 209 209 209
20	210 210 210 210 210
	211 211 211 211 211
	212 212 212 212 212
	213 213 213 213 213
	214 214 214 214 214
	215 215 215 215 215
25	216 216 216 216 216
	217 217 217 217 217
	218 218 218 218 218
	219 219 219 219 219
	220 220 220 220 220
30	221 221 221 221 221
	222 222 222 222 222
	223 223 223 223 223
	224 224 224 224 224
	225 225 225 225 225
35	226 226 226 226 226
	227 227 227 227 227
	228 228 228 228 228
	229 229 229 229 229
	230 230 230 230 230
	231 231 231 231 231
40	232 232 232 232 232
	233 233 233 233 233
	234 234 234 234 234
	235 235 235 235 235
	236 236 236 236 236
45	237 237 237 237 237
	238 238 238 238 238
	239 239 239 239 239
	240 240 240 240 240
	241 241 241 241 241
50	242 242 242 242 242
	243 243 243 243 243
	244 244 244 244 244
	245 245 245 245 245

```

246 246 246 246 246
247 247 247 247 247
5   248 248 248 248 248
249 249 249 249 249
250 250 250 250 250
251 251 251 251 251
252 252 252 252 252
253 253 253 253 253
10  254 254 254 254 254
255 255 255 255 255
:::::::::::
cube4/main.C
:::::::::::
// main.C
15 // (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

20 #include "Test.h"

main(int argc, char *argv[]) { Test::Run(argc,argv); } // main

// end of main.C
:::::::::::
25 oop/DynaArray/DynaArray.C
:::::::::::
// DynaArray.C
// (c) Ingmar Bitter '97

// Copyright, Mitsubishi Electric Information Technology Center
30 // America, Inc., 1997, All rights reserved.

#include "DynaArray.h"

35 template <class T> void DynaArray<T>::Demo()
{
    DynaArray<int> numbers;
    cout << endl << "Demo of class " << typeid(numbers).name();
    cout << endl << "size : " << sizeof(DynaArray<T>) << " Bytes";
    cout << endl << "public member functions:" ;
40    cout << endl << " DynaArray<int> numbers: numbers = " << numbers << &numbers;
    cout << endl << " numbers.Size() : " << numbers.Size() << "
numbers = "<< numbers;
    cout << endl << " numbers.SetSize(5) : " << numbers.SetSize(5) << "
numbers = "<< numbers;
    cout << endl << " numbers.Size() : " << numbers.Size() << "
45    numbers = "<< numbers;
    cout << endl << " numbers[0] = 4 : " << (numbers[0] = 4) << "
numbers = "<< numbers;
    cout << endl << " numbers[1] = 6 : " << (numbers[1] = 6) << "
numbers = "<< numbers;
50

```

```

367
    cout << endl << " numbers.SetSize(2) : " << numbers.SetSize(2) << " numbers
= "<< numbers;
5     DynaArray<int> nums(numbers);
    cout << endl << " DynaArray<int> nums(numbers): nums = " << nums << &nums;
    cout << endl << " numbers.Flush() : " << numbers.Flush() << "
numbers = "<< numbers;
    cout << endl << " numbers(nums) : " << numbers(nums) << " numbers
= "<< numbers;
    cout << endl << " numbers.Flush() : " << numbers.Flush() << "
numbers = "<< numbers;
10   cout << endl << " numbers=nums : " << (numbers=nums) << " numbers
= "<< numbers;

    cout << endl << "End of demo of class " << typeid(numbers).name() << endl;
15 } // Demo

///////////////////////////////
// constructors & destructors

20
template <class T> DynaArray<T>::DynaArray(unsigned int setSize)
    : size(setSize), space(setSize)
{
25     // make space for elements
    elements = new T[space];
    assert(elements);

    // set all elements to 0
    Flush();
}

30 } // constructor

template <class T> DynaArray<T>::DynaArray(ccnst DynaArray<T> & source)
    : size(source.size), space(source.size)
{
35     // make space for elements
    elements = new T[size];
    assert(elements);

    // set each element to the initial value
    for (unsigned int k = 0; k<size; ++k)
40        elements[k] = source.elements[k];
}

45 } // copy-constructor

template <class T> DynaArray<T>::~DynaArray()
{
    // free datastructures
    assert(elements);
    delete [] elements;
}

```

50

55

```

368
elements = 0;

5 } // destructor

///////////////////////////////
// show/set data & data properties
//
10 // - class Object requirements

template <class T> ostream & DynaArray<T>::Ostream(ostream & os) const
{
15    // append array info to os
    os << "(";
    if (!size)
        os << "empty DynaArray";
    else
20    os << elements[0];
    for (unsigned int k=1; k<size; ++k) {
        os << "," << elements[k];
    }
    os << ")";
25    // return complete os
    return os;
}

30 } // Ostream

///////////////////////////////
// show/set data & data properties
//
// - local show/set functions

35 template <class T> unsigned int DynaArray<T>::Size() const
{
    return size;
} // Size

40 template <class T> unsigned int DynaArray<T>::SetSize (unsigned int setSize)
{
    // dynamic change of array size
    unsigned int k;

45    if (space/4 < setSize && setSize <= space) {
        size = setSize;
        return size;
    }

50    space = 2*setSize;
}

```

55

```

369
// allocate memory for new array
T * newElements = new T[space];
assert(newElements);

5   if ( space < size ) {

        // copy the first setSize elements into smaller array
        for (k=0; k<space; ++k)
            newElements[k] = elements[k];
    }
10  else {
        // space > size

        // copy all size elements into bigger array
        for (k=0; k<size; ++k)
            newElements[k] = elements[k];

        // initialize the remaining ones to 0
        for (k=size; k<space; ++k)
            newElements[k] = 0;
    }

20  // delete the old array
assert(elements);
25  delete [] elements;

        // update the data member fields
        size = setSize;
        elements = newElements;

30  // return new size
return size;

} // SetSize

35
template <class T> DynaArray<T> & DynaArray<T>::Flush()
{
    // set each element to 0
    for (unsigned int k = 0; k<space; ++k)
        elements[k] = 0;

    return *this;
} // Flush

45
template <class T> T & DynaArray<T>::operator [] (unsigned int index) const
{
    // subscript a array value
    // check that index is valid
    assert(index < size);

50  // return requested element

```

55

370

```

    return elements[index];

5   } // operator []

template <class T> DynaArray<T> & DynaArray<T>::operator =
  (const DynaArray<T> & source)
{
10  // match sizes
  if (size != source.size) {
    // allocate memory for new array
    space = source.size;
    T * newElements = new T[space];
    assert(newElements);

15  // delete the old array
    assert(elements);
    delete [] elements;

20  // update the data member fields
    size = space;
    elements = newElements;
  }

25  // copy the elements
  for (unsigned int k=0; k<size; ++k)
    elements[k] = source.elements[k];

30  return *this;
} // operator =

template <class T> DynaArray<T> & DynaArray<T>::operator ()
  (const DynaArray<T> & source)
{
35  *this = source;

  return *this;
} // operator ()

40

// end of DynaArray.C
:::::::::::
oop/DynaArray/DynaArray.h
:::::::::::
// DynaArray.h
// (c) Ingmar Bitter '97

45

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

50

// dynamic array ADT

```

55

```

371
#ifndef _DynaArray_h_ // prevent      multiple includes
#define _DynaArray_h_
5
#include "assert.h"
#include "Object.h"

10 template <class T> class DynaArray : public virtual Object {
public:
    static void           Demo          ();
    // constructors & destructors
    DynaArray   (unsigned int setSize=0);
    DynaArray   (const DynaArray<T> &);
15   virtual ~DynaArray  ();
    // show/set data & data properties
    // - class Object requirements
    virtual ostream &  Ostream (ostream & ) const;
20   virtual /*inline*/ unsigned int   Size      ( ) const;
    virtual         unsigned int   SetSize     (unsigned int
setSize);
    virtual /*inline*/ DynaArray<T> &  Flush      ( ); // set all
elements to 0
25   virtual /*inline*/ T &          operator [] (unsigned int index)
const;
    virtual         DynaArray<T> & operator = (const DynaArray<T> &);
    virtual         DynaArray<T> & operator () (const DynaArray<T> &);

30 protected:
    unsigned int size; // reported bye Size()
    unsigned int space; // realy allocated memory
    T * elements;

35 }; // class DynaArray
#endif // _DynaArray_h_
:::::::::::
oop/DynaArray/Makefile
:::::::::::
40 #   Makefile for C++ programs (c) Ingmar Bitter '97
#
#   make             : compile only changed files and their depend
files
#   make all          : recompile all
#   make clean        : recompile all and remove unnecessary files
45 #
EXECUTABLE = go

SRC    = main.C Test.C Object.C DynaArray.C
50

```

372

```

RM = /bin/rm -fs

5      #
#      put -g here | to add debugging info to executable
#          V
CDEBUGFLAGS = -O3
CCOPTIONS   = -n32

10     CC = CC

INCLUDES    =
LIBS        =
CFLAGS       = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

15     OFILES = $(SRC:.C=.o)

$(EXECUTABLE): $(OFILES)
                $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

20     #
#      target: dependency \n tab rule
#
.C.o:
        $(CC) $(CFLAGS) -c $<

25     #
#      switch:: ; \n tab rule
#
clean:    ;
        find . -name "*.o" -print -exec mv {} ~/dumpster \; ;
        find . -name "*~"  -print -exec mv {} ~/dumpster \; ;
        find . -name "go"   -print -exec mv {} ~/dumpster \; ;
        find . -name "core" -print -exec mv {} ~/dumpster \; ;

all:    ;
        pmake -u

35     depend:
        makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

40     main.o: Test.h DynaArray.h /usr/include/assert.h Object.h
main.o: /usr/include/string.h /usr/include/standards.h Global.h
main.o: /usr/include/stdlib.h /usr/include/sgidefs.h
Test.o:  Test.h DynaArray.h /usr/include/assert.h Object.h
Test.o: /usr/include/string.h /usr/include/standards.h Global.h
Test.o: /usr/include/stdlib.h /usr/include/sgidefs.h
45     Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h /usr/include/stdlib.h /usr/include/sgidefs.h
DynaArray.o: DynaArray.h /usr/include/assert.h Object.h /usr/include/string.h
DynaArray.o: /usr/include/standards.h Global.h /usr/include/stdlib.h
DynaArray.o: /usr/include/sgidefs.h
50

```

55

```

::::::::::
oop/DynaArray/Test.C
::::::::::
// Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

#include "Test.h"

10 void Test::Run()          { DynaArray<int>::Demo(); }

// end of Test.cpp
::::::::::
oop/DynaArray/Test.h
::::::::::
// Test.h        test class for OOP-classes
// (c) Ingmar Bitter '96

20 #ifndef _Test_h_           // prevent multiple includes
#define _Test_h_

#include "DynaArray.h"

class Test { public: static void Run(); };

25 #endif      // _Test_h_
::::::::::
oop/FiFo/FiFo.C
::::::::::
// FiFo.C
30 // (c) Ingmar Bitter '97 / Urs Kanus '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

35 #include "FiFo.h"
#include "ModInt.h"

template <class T> void FiFo<T>::Demo()
{
40     FiFo<T> fifo;
    T a(1),b(2),c(3),d(4),e(5),f(6),out;
    cout << endl << "Demo of class " << typeid(fifo).name();
    cout << endl << "size : " << sizeof(FiFo<T>) << " Bytes";
    cout << endl << "public member functions:";

45     cout << endl << "  Size() => " << fifo.Size();
    cout << endl << Object::BoolStr[fifo.Size()==1];
    cout << endl << "SetSize("<<c<<") "; fifo.SetSize(3);
    cout << endl << Object::BoolStr[fifo.Size()==3];
    cout << endl << "Preset("<<a<<") => "; fifo.Preset(a); cout << fifo;
50     cout << endl << Object::BoolStr[fifo.mem[0]==a && fifo.mem[1]==a &&
    fifo.mem[2]==a];
}

```

374

```

cout << "Read(out); => ";           fifo.Read(out);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==a];
cout << "Read(out); => ";   fifo.Read(out);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==a];
cout << "Read(out); => ";   fifo.Read(out);
cout << "out=<<out<<"; fifo=<<fifo;
10   cout << endl <<Object::BoolStr[out==a];

cout << "Write("<<b<<") => ";   fifo.Write(b); cout<<"fifo=<<fifo;
cout << endl <<Object::BoolStr[fifo.mem[0]==b && fifo.mem[1]==a &&
15   fifo.mem[2]==a];
cout << "Write("<<c<<") => ";   fifo.Write(c); cout<<"fifo=<<fifo;
cout << endl <<Object::BoolStr[fifo.mem[0]==b && fifo.mem[1]==c &&
fifo.mem[2]==a];
cout << "Write("<<d<<") => ";   fifo.Write(d); cout<<"fifo=<<fifo;
20   cout << endl <<Object::BoolStr[fifo.mem[0]==b && fifo.mem[1]==c &&
fifo.mem[2]==d];

cout << "Peek(0); => "; out = fifo.Peek(0);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==b];
25   cout << "Peek(1); => "; out = fifo.Peek(1);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==c];
cout << "Peek(2); => "; out = fifo.Peek(2);
cout << "out=<<out<<"; fifo=<<fifo;
30   cout << endl <<Object::BoolStr[out==d];
cout << "Peek(-1); => "; out = fifo.Peek(-1);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==d];

cout << "Read(); => "; out = fifo.Read();
35   cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==b];
cout << "Read(); => "; out = fifo.Read();
cout << "out=<<out<<"; fifo=<<fifo;
40   cout << endl <<Object::BoolStr[out==c];
cout << "Read(); => "; out = fifo.Read();
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==d];

cout << "Exchange("<<e<<",out) => ";   fifo.Exchange(e,out);
45   cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==b];
cout << "Exchange("<<f<<",out) => ";   fifo.Exchange(f,out);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==c];

50   cout << "Read(out); => ";   fifo.Read(out);
cout << "out=<<out<<"; fifo=<<fifo;
cout << endl <<Object::BoolStr[out==d];

```

55

```

      375
    cout << endl << "End of demo of class " << typeid(fifo).name() << endl;
} // Demo
5

///////////////////////////////
// constructors & destructors

10 template <class T> FiFo<T>::FiFo(int setSize)
    : size(setSize), writePointer(0), readPointer(0), fifoFull(false)
{
    mem = new T [size];
15    if (!mem) ERROR("OutOfMemoryError");
} // constructor

20 template <class T> FiFo<T>::~FiFo()
{
    if (mem) { delete mem; mem = 0; }
} // destructor

25 /////////////////////////
// show/set data & data properties
//
// - class Object requirements

30 template <class T> ostream & FiFo<T>::Ostream(ostream & os) const
{
    // append FiFo info to os
    os << "(writePointer=" << writePointer;
    os << ", readPointer=" << readPointer;
/*
    << "; data=[";
35    for (int k=0; k<size-1; ++k) {
        os << &mem[k] << ", ";
    }
    os << &mem[size-1] << "]"
*/
    os << "}";
40    os << "}";

    // return complete os
    return os;
}

45    // Ostream
}

///////////////////////////////
// show/set data & data properties
//
50    // - local show/set functions

```

```

376
template <class T> bool           FiFo<T>::SetSize(const int& setSize)
{
    size = setSize;
    if (mem) delete mem;
    mem = new T [size];
    if (!mem) ERROR("OutOfMemoryError");

    writePointer = readPointer = 0;
    fifoFull = false;

    return true;
} // SetSize

15 template <class T> int FiFo<T>::Size() const
{
    return size;
} // Size

20 template <class T> void FiFo<T>::Preset(const T& value )
{
    if (size>0) {
        for (int i=0; i<size; i++)
            mem[i] = value;
    }
    writePointer = readPointer = 0;
    fifoFull = false;
}
/////////////////////////////// /////////////////////////////////
// local computation functions

30 template <class T> bool FiFo<T>::Write( const T& in )
{
    if ( fifoFull )
        ERROR( "FiFo is full" );

    mem[writePointer] = in;
    writePointer = (writePointer + 1) % size;

    if ( writePointer == readPointer )
        fifoFull = true;

    return true;
} // Write

40 template <class T> bool FiFo<T>::Read(T& out)
{
    if ( fifoFull ) fifoFull = false;

    out = mem[readPointer];
    readPointer = (readPointer + 1) % size;

    return true;
}

```

377

```

} // Read

5   template <class T> T FiFo<T>::Read()
{
    T out;

    if ( fifoFull ) fifoFull = false;
10
    out = mem[readPointer];
    readPointer = (readPointer + 1) % size;

    return out;
} // Read
15

template <class T> T FiFo<T>::Peek( const int offsetFromReadPointer )
{
    T out;
    ModInt peekPointer(0, size);
20
    peekPointer = readPointer + offsetFromReadPointer;
    out = mem[peekPointer];

    return out;
} // Read
25

template <class T> bool FiFo<T>::Exchange( const T& in, T& out)
{
    if (size>0) {
        Read(out);
        Write(in);
    }
    else {
        out=in;
    }

    return true;
} // Exchange
30
35
40
///////////
// internal utility functions

45
// end of FiFo.C
:::::::::::
oop/FiFo/FiFo.h
:::::::::::
// FiFo.h
// (c) Ingmar Bitter '97 / Urs Kanus '97
50
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

```

55

```

// FiFo (Write, Read, Exchange)
5
#ifndef _FiFo_h_ // prevent multiple includes
#define _FiFo_h_

#include "Object.h"

10 template <class T> class FiFo : virtual public Object {
public:

    static void Demo ();

15    // constructors & destructors
    FiFo(int setSize=1);
    virtual ~FiFo();

    // show/set data & data properties
20    // - class Object requirements
    virtual ostream & Ostream (ostream & ) const;

    // - local show/set functions
    virtual bool SetSize( const int& setSize );
    virtual int Size() const;
25    virtual void Preset( const T& value );

    // local computation functions
    virtual bool Write( const T& in );
    virtual bool Read ( T& out );
30    virtual T Read ();
    virtual T Peek ( const int offsetFromReadPointer );
    virtual bool Exchange( const T& in, T& out );

protected:
35    T *mem;
    int size, writePointer, readPointer;
    bool fifoFull;
};

#endif // _FiFo_h_
40 ::::::::::::::::::::
oop/Fifo/Makefile
::::::::::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
#          make                         : compile only changed files and their depend
45 files
#      make all           : recompile all
#      make clean        : recompile all and remove unnecessary files
#
50 EXECUTABLE = go

```

379

```

SRC    = main.C Test.C Object.C Fifo.C      ModInt.C

5      RM = /bin/rm -fs

#
#      put -g here | to add debugging info to executable
#          V
CDEBUGFLAGS = -g
10     CCOPTIONS = -n32

CC = CC

INCLUDES =
LIBS =
15    CFLAGS      = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

OFILES = $(SRC:.C=.o)

$(EXECUTABLE): $(OFILES)
20      $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

#
#      target: dependency \n tab rule
#
.C.o:
25      $(CC) $(CFLAGS) -c $<

#
#      switch:: ; \n tab rule
#
clean: ;
30      find . -name "*.o" -print -exec mv {} ~/dumpster \; ;
      find . -name "*~"  -print -exec mv {} ~/dumpster \; ;
      find . -name "go"   -print -exec mv {} ~/dumpster \; ;
      find . -name "core" -print -exec mv {} ~/dumpster \; ;

all: ;
35      pmake -u

depend:
        makedepend -- $(CFLAGS) -- $(SRC)

40      # DO NOT DELETE

main.o: Test.h Fifo.h Object.h /usr/include/string.h /usr/include/standards.h
main.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
main.o: /usr/include/sgidefs.h
Test.o: Test.h Fifo.h Object.h /usr/include/string.h /usr/include/standards.h
45      Test.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
Test.o: /usr/include/sgidefs.h
Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h /usr/include/stdlib.h /usr/include/sgidefs.h
Fifo.o: Fifo.h Object.h /usr/include/string.h /usr/include/standards.h

```

50

380

```

FiFo.o: Global.h /usr/include/assert.h    /usr/include/stdlib.h
FiFo.o: /usr/include/sgidefs.h ModInt.h /usr/include/limits.h
ModInt.o: ModInt.h Object.h /usr/include/string.h /usr/include/standards.h
ModInt.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
ModInt.o: /usr/include/sgidefs.h /usr/include/limits.h
:::::::::::
oop/FiFo/Test.C
:::::::::::
10 // Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

#include "Test.h"

15 void Test::Run() { FiFo<int>::Demo(); }

// end of Test.cpp
:::::::::::
oop/FiFo/Test.h
:::::::::::
20 // Test.h          test class for OOP-classes
// (c) Ingmar Bitter '96

#ifndef _Test_h_           // prevent multiple includes
#define _Test_h_
25
#include "FiFo.h"

class Test { public: static void Run(); };

30 #endif      // _Test_h_
:::::::::::
oop/FixPointNumber/FixPointNumber.C
:::::::::::
// FixPointNumber.cpp      class FixPointNumber
// (c) Ingmar Bitter '96 / Urs Kanus '97
35

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

40 #include "FixPointNumber.h"
char FixPointNumber::defaultFractionBits = 8;
char FixPointNumber::maxFractionBits = 32;

45 void FixPointNumber::Demo()
{
    double a(0.5);

    FixPointNumber k(2.5), m(k), n(k*k), p;
    FixPointNumber kp(2, 8), mp(kp), np(kp*kp);
50    FixPointNumber pp(2.5, 8);

```

55

```

381
cout << endl << "Demo of class " << typeid(k).name();
cout << endl << "size : " << sizeof(FixPointNumber) << " Bytes";
cout << endl << "public members:";
5   cout << endl << "public member functions:";
cout << endl << " double a(0.5) = "<<a;
cout << endl << " FixPointNumber k(2.5), m(k), n(k*k), p = "<<k<<, "<<m<<,
"<<n<<, "<<p;
cout << endl << "\t"<<Object::BoolStr[k==2.5 && m==2.5 && n==6.25 && p==0];
10  cout << " k.FractionBits() = "<<(int)(k.FractionBits());
cout << endl
<<"\t"<<Object::BoolStr[(int)(k.FractionBits())]==defaultFractionBits];
SetDefaultFractionBits(8); FixPointNumber test(2);
cout << " SetDefaultFractionBits(8); FixPointNumber test(2);
test.FractionBits() = "<<(int)(test.FractionBits());
15  cout << endl << "\t"<<Object::BoolStr[(int)(test.FractionBits())]==8];
cout << " FixPointNumber kp(2, 8), mp(kp), np(kp*kp), pp = "<<kp<<, "<<mp<<,
"<<np<<, "<<pp;
cout << endl << "\t"<<Object::BoolStr[kp==2 && mp==2 && np==4 && pp==2.5];
20  cout << endl << " p(-8) = "<<p(-8);
cout << endl << "\t"<<Object::BoolStr[p== -8];
cout << endl << " k.Abs() = "<< k.Abs()<<"\tp.Abs() = "<< (p.Abs());
cout << endl << "\t"<<Object::BoolStr[k.Abs() == 2.5 && p.Abs() == 8];
25  cout << " k==k = "<<(k==k)<<"\tk<<k = "<<(k<=k)<<"\tk<<k = "<<(k<k)
"<<(k<=n);
cout << endl << "\t"<<Object::BoolStr[(k==k)==1 && (k<=k)==1 && (k<k)==0 &&
(k==9)==0 && (9.<=k)==0 && (k<n)==1];
30  cout << " k!=k = "<<(k!=k)<<"\tk>=k = "<<(k>=k)<<"\tk>k = "<<(k>k)
"<<"\tk!=9 = "<<(k!=9)<<"\t9.>=k = "<<(9.>=k)<<"\tk>n =
"<<(k>n);
cout << endl << "\t"<<Object::BoolStr[(k!=k)==0 && (k>=k)==1 && (k>k)==0 &&
(k!=9)==1 && (9.>=k)==1 && (k>n)==0];
35  cout << " ++k = "<<(++k) <<"\tn+m = "<<(n+m);
cout << "\tm+a = "<<(m+a) <<"\t\tp+a = "<<(p+=a);
cout << endl << "\t"<<Object::BoolStr[k==3.5 && (n+m)==8.75 && (m+a)==3 &&
p== -7.5];
40  cout << " k++ = "<<(k++) <<"\tn-m = "<<(n-m);
cout << "\tm-a = "<<(m-a) <<"\t\tp-a = "<<(p-=a);
cout << endl << "\t"<<Object::BoolStr[k==4.5 && (n-m)==3.75 && (m-a)==2 &&
p== -8];
45  cout << " --k = "<<(--k) <<"\tn*m = "<<(n*m);
cout << "\tm*a = "<<(m*a) <<"\t\tp*a = "<<(p*=a);
cout << endl << "\t"<<Object::BoolStr[k==3.5 && (n*m)==15.625 &&
(m*a)==1.25 && p== -4];
50  cout << " k-- = "<<(k--) <<"\tn/m = "<<(n/m);

```

50

55

382

```

cout << "\tm/a =
      cout << endl << "\t" << Object::BoolStr[k==2.5 && (n/m)==2.5 && (m/a)==5 &&
5   p==8];

cout << " k+kp = <<(k+kp) <<"\t k-kp = <<(k-kp);
cout << "\t k*kp = <<(k*kp) <<"\t k/kp = <<(k/kp);
cout << endl << "\t" << Object::BoolStr[(k+kp)==4.5 && (k-kp)==0.5 && (k*kp)==5
&& (k/kp)==1.25];
10   cout << " k.Abs() = << k.Abs()<<"\tp.Abs() = << (p.Abs());
      cout << endl << "\t" << Object::BoolStr[k.Abs()]==2.5 && p.Abs()==8];

cout << " kp==kp = <<(kp==kp)<<"\tkp<=kp = <<(kp<=kp)<<"\tkp>kp = <<(kp<kp)
<<"\tkp<=9 = <<(kp==9)<<"\t9.<=kp = <<(9.<=kp)<<"\tkp<np =
15   " <<(kp<np);
      cout << endl << "\t" << Object::BoolStr[(kp==kp)==1 && (kp<=kp)==1 &&
(kp<kp)==0 && (kp==9)==0 && (9.<=kp)==1 && (kp<np)==1];

cout << " kp!=kp = <<(kp!=kp)<<"\tkp>=kp = <<(kp>=kp)<<"\tkp>kp = <<(kp>kp)
<<"\tkp!=9 = <<(kp!=9)<<"\t9.>=kp = <<(9.>=kp)<<"\tkp>np =
20   " <<(kp>np);
      cout << endl << "\t" << Object::BoolStr[(kp!=kp)==0 && (kp>=kp)==1 &&
(kp>kp)==0 && (kp!=9)==1 && (9.>=kp)==1 && (kp>np)==0];

cout << " ++kp = <<(++kp) <<"\tnp+mp = <<(np+mp);
25   cout << "\tmp+a = <<(mp+a) <<"\t\tpp+=a = <<(pp+=a);
      cout << endl << "\t" << Object::BoolStr[kp==3 && (np+mp)==6 && (mp+a)==2.5 &&
pp==3];

cout << " kp++ = <<(kp++) <<"\tnp-mp = <<(np-mp);
20   cout << "\tmp-a = <<(mp-a) <<"\t\tpp-=a = <<(pp-=a);
      cout << endl << "\t" << Object::BoolStr[kp==4 && (np-mp)==2 && (mp-a)==1.5 &&
pp==2.5];
30   cout << endl << "\t" << Object::BoolStr[kp==3 && (np*mp)==8 && (mp*a)==1 &&
pp==1.25];

cout << " --kp = <<(--kp) <<"\tnp*mp = <<(np*mp);
35   cout << "\tmp*a = <<(mp*a) <<"\t\tpp*=a = <<(pp*=a);
      cout << endl << "\t" << Object::BoolStr[kp==3 && (np*mp)==8 && (mp*a)==1 &&
pp==1.25];

cout << " kp-- = <<(kp--) <<"\tnp/mp = <<(np/mp);
40   cout << "\tmp/a = <<(mp/a) <<"\t\tpp/=a = <<(pp/=a);
      cout << endl << "\t" << Object::BoolStr[kp==2 && (np/mp)==2 && (mp/a)==4 &&
pp==2.5];

cout << " kp+=k = <<(kp+=k) <<"\tkp+=kp = <<(k+=kp);
45   cout << endl << "\t" << Object::BoolStr[kp==4.5 && k==7];

FixPointNumber round1(0.4,16), round2(0.5,16), round3(0.6,16);

cout << " round1(0.4,16) = <<round1<<"\tround2(0.5,16) =
" <<round2<<"\tround3(0.6,16) = <<round3;
50   cout << endl << "\t" << Object::BoolStr[round1==round1 && round2>=round? &&
round3>=round3];

```

50

55

```

383
    cout << " round1.Round(0) =
5   "<<round1.Round(0)<<"\t\tround2.Round(0) =
    "<<round2.Round(0)<<"\tround3.Round(0) = "<<round3.Round(0);
    cout << endl <<"\t"<<Object::BoolStr[round1.Round(0)==0 && round2.Round(0)==1
&& round3.Round(0)==1];
    cout << " round1.Round(1) = "<<round1.Round(1)<<"\t\tround2.Round(1) =
    "<<round2.Round(1)<<"\tround3.Round(1) = "<<round3.Round(1);
    cout << endl <<"\t"<<Object::BoolStr[round1.Round(1)==0.5 &&
10  round2.Round(1)==0.5 && round3.Round(1)==0.5];
    cout << " round1.Round(2) = "<<round1.Round(2)<<"\t\tround2.Round(2) =
    "<<round2.Round(2)<<"\tround3.Round(3) = "<<round3.Round(2);
    cout << endl <<"\t"<<Object::BoolStr[round1.Round(2)==0.5 &&
round2.Round(2)==0.5 && round3.Round(2)==0.5];
    cout << " round1.Round(3) = "<<round1.Round(3)<<"\t\tround2.Round(3) =
15  "<<round2.Round(3)<<"\tround3.Round(3) = "<<round3.Round(3);
    cout << endl <<"\t"<<Object::BoolStr[round1.Round(3)==0.375 &&
round2.Round(3)==0.5 && round3.Round(3)==0.625];

    FixPointNumber s( 255, 8);
20   cout << " FixPointNumber s( 255, 8); s>>8 = "<<(s>>8);
    cout << endl <<"\t"<<Object::BoolStr[((s>>8) - 0.996094) < 0.00001];
    cout << " s<<8 = "<<(s<<8);
    cout << endl <<"\t"<<Object::BoolStr[(s<<8)==65280];

    FixPointNumber doubleToFixP(0.847656 , 12);
25   cout << endl << " FixPointNumber doubleToFixP(0.847656 , 12) =
    "<<doubleToFixP;
    cout << endl << " double(doubleToFixP) = "<<double(doubleToFixP);
    FixPointNumber doubleToFix2(double(doubleToFixP), 8);
    cout << endl << " FixPointNumber doubleToFix2(double(doubleToFixP), 8) =";
    cout <<doubleToFix2;
30   cout << endl <<"End of demo of class " << typeid(k).name() << endl;
}      // Demo

////////////////////////////////////////////////////////////////
35 // constructors & destructors

    FixPointNumber::FixPointNumber      (const int k)
        : n(((FixPointData) k)<<defaultFractionBits),
fractionBits(defaultFractionBits) { }

40   FixPointNumber::FixPointNumber      (const int k, const char p)
        : n(((FixPointData) k)<<p), fractionBits(p) { }

    FixPointNumber::FixPointNumber      (const FixPointData k)
        : n(k<<defaultFractionBits), fractionBits(defaultFractionBits) { }

45   FixPointNumber::FixPointNumber      (const double x)
        : n((FixPointData) (x * (1<<defaultFractionBits))), fractionBits(defaultFractionBits) { }

    FixPointNumber::FixPointNumber      (const double x, const char p)
50

```

```

      384
    : n((FixPointData) (x * (1<<p))), fractionBits(p) { }

5   FixPointNumber::FixPointNumber      (const FixPointNumber & k)
    : n(k.n), fractionBits(k.fractionBits) { }

10
// show/set data & data properties
//
// - class Object requirements

15   ostream &   FixPointNumber::Ostream (ostream & os) const
    // { return os << double(*this) << ":" << int(fractionBits); }
{ return os << double(*this); }

20
// casts to other data types

25   FixPointNumber::operator double () const { return double(n) / (1 <<
fractionBits); }

//
// show/set data & data properties
//
// - class Object requirements

30   double FixPointNumber::Norm () const { return double(*this); }

35   void FixPointNumber::SetDefaultFractionBits( const char p )
{ defaultFractionBits = p; }

FixPointNumber FixPointNumber::Round ( const char p ) const
{
  FixPointNumber temp;
  int fractionBitsDiff = fractionBits - p;
  bool round = false;
  int cutoffBit;
  if (fractionBitsDiff > 0) {
    cutoffBit = 1<<(fractionBitsDiff-1);
    round = ((cutoffBit & n) == cutoffBit);
    temp.n = (n>>fractionBitsDiff)+(int)round;
    temp.fractionBits = p;
  }
  else {
    temp.n = n<<fractionBitsDiff;
    temp.fractionBits = p;
}

```

385

```

    }

5     return temp;
}

FixPointNumber FixPointNumber::Abs ( ) const
{ if (n<0) return FixPointNumber(((double)(-n))/(1<<fractionBits),
fractionBits); else return *this; }

10    FixPointNumber & FixPointNumber::operator ()      (const int k)
{
    n=(k<<defaultFractionBits);
    fractionBits=defaultFractionBits;
    return *this;
}

15    FixPointNumber & FixPointNumber::operator ()      (const double x)
{
    n=(FixPointData) (x * (1<<defaultFractionBits));
    fractionBits=defaultFractionBits;
    return *this;
}

20    }

FixPointNumber   FixPointNumber::operator +      ( ) const
{ return FixPointNumber(n); }
FixPointNumber   FixPointNumber::operator -      ( ) const
{ return FixPointNumber(-double(*this)); }

25    FixPointNumber & FixPointNumber::operator '++'    ( ) { n += (1<<fractionBits);
return *this; }
FixPointNumber & FixPointNumber::operator ++    (int) { n += (1<<fractionBits);
return *this; }
FixPointNumber & FixPointNumber::operator --    ( ) { n -= (1<<fractionBits);
return *this; }
FixPointNumber & FixPointNumber::operator --    (int) { n -= (1<<fractionBits);
return *this; }

30    // change +=, -=, *=, /=
FixPointNumber & FixPointNumber::operator +=     (const FixPointNumber& k)
{
    int fractionBitsDiff = fractionBits - k.fractionBits;

    if (fractionBitsDiff > 0)
        n += (k.n<<fractionBitsDiff);
    else if (fractionBitsDiff < 0)
        n += (k.n>>(-fractionBitsDiff));
    else n += k.n;
    return *this;
}

40    FixPointNumber & FixPointNumber::operator -=     (const FixPointNumber& k)
{ n -= k.n; return *this; }
FixPointNumber & FixPointNumber::operator *=     (const FixPointNumber& k)
{ n=FixPointNumber(double(*this) * double(k)).n; return *this; }
FixPointNumber & FixPointNumber::operator /=     (const FixPointNumber& k)

45
}

```

50

55

```

386
{ n=FixPointNumber(double(*this) /      double(k)).n; return *this; }

5   FixPointNumber & FixPointNumber::operator +=  (const double& k)
{ n += FixPointNumber(k, fractionBits).n; return *this; }
FixPointNumber & FixPointNumber::operator -=  (const double& k)
{ n -= FixPointNumber(k, fractionBits).n; return *this; }
FixPointNumber & FixPointNumber::operator *=  (const double& k)
{ n=FixPointNumber(double(*this) * k, fractionBits).n; return *this; }
10  FixPointNumber & FixPointNumber::operator /=  (const double& k)
{ n=FixPointNumber(double(*this) / k, fractionBits).n; return *this; }

FixPointNumber  FixPointNumber::operator +      (const FixPointNumber& k) const
{
15    char resultFractionBits;
    if (fractionBits >= k.fractionBits)
        resultFractionBits = fractionBits;
    else
        resultFractionBits = k.fractionBits;

20    return FixPointNumber(double(*this) + double(k), resultFractionBits);
}
FixPointNumber  FixPointNumber::operator -      (const FixPointNumber& k) const
{
    char resultFractionBits;
25    if (fractionBits >= k.fractionBits)
        resultFractionBits = fractionBits;
    else
        resultFractionBits = k.fractionBits;

30    return FixPointNumber(double(*this) - double(k), resultFractionBits);
}

FixPointNumber  FixPointNumber::operator *      (const FixPointNumber& k) const
{
35    char resultFractionBits = fractionBits + k.fractionBits;
    if (resultFractionBits > maxFractionBits)
        return FixPointNumber(double(*this) * double(k), maxFractionBits);
    else
        return FixPointNumber(double(*this) * double(k),
resultFractionBits);
40
}

FixPointNumber  FixPointNumber::operator /      (const FixPointNumber& k) const
{
45    char resultFractionBits;
    if (fractionBits >= k.fractionBits)
        resultFractionBits = fractionBits;
    else
        resultFractionBits = k.fractionBits;
}

```

50

55

387

```

        return FixPointNumber(double(*this) / double(k), resultFractionBits);
    }

    FixPointNumber FixPointNumber::operator +      (const double& k) const
    {
        return FixPointNumber(double(*this) + k);
    }

    FixPointNumber FixPointNumber::operator -      (const double& k) const
    {
        return FixPointNumber(double(*this) - k);
    }

    FixPointNumber FixPointNumber::operator *      (const double& k) const
    {
        return FixPointNumber(double(*this) * k);
    }

    FixPointNumber FixPointNumber::operator /      (const double& k) const
    {
        return FixPointNumber(double(*this) / k);
    }

    int FixPointNumber::operator &      (const int k) const
    { FixPointNumber temp(*this); return (temp.n & k); }

    FixPointNumber FixPointNumber::operator <<   (const int k) const
    { FixPointNumber temp(*this); temp.n = temp.n<<k; return temp; }

    FixPointNumber FixPointNumber::operator >>   (const int k) const
    { FixPointNumber temp(*this); temp.n = temp.n>>k; return temp; }

    char          FixPointNumber::FractionBits( ) const { return fractionBits; }

// end of FixPointNumber

:::::::::::
35  oop/FixPointNumber/FixPointNumber.h
:::::::::::
// FixPointNumber.h           class FixPointNumber
// (c) Ingmar Bitter '96 / Urs Kanus '97

// Copyright, Mitsubishi Electric Information Technology Center
40  // America, Inc., 1997, All rights reserved.

// Policy for determining precision of division results:
//   Use precision of the higher precision operand

45  #ifndef _FixPointNumber_h_           // prevent multiple includes
#define _FixPointNumber_h_

#include "Object.h"
typedef long FixPointData;

```

50

55

```

5      class FixPointNumber : virtual public Object {
public:
    virtual void MoveRight() { n = n >> fractionBits; }

10     static void      Demo ();

// constructors & destructors
FixPointNumber (const int k);
FixPointNumber (const int k, const char p);
FixPointNumber (const FixPointData k=0);
FixPointNumber (const double x);
FixPointNumber (const double x, const char p);
15     FixPointNumber (const FixPointNumber & k);

        // casts to other data types
        virtual /*inline*/ operator double () const;

20     // show/set data & data properties
        // - class Object requirements
        virtual          ostream &   Ostream (ostream & ) const;

        // - class CmpObj requirements
virtual double      Norm      ( )           const;
25     // - local show/set functions

/*inline*/ static void SetDefaultFractionBits(const char p );

30     /*inline*/ virtual FixPointNumber Round (const char p) const;
/*inline*/ virtual FixPointNumber Abs ( ) const;

/*inline*/ virtual FixPointNumber & operator () (const int      k);
/*inline*/ virtual FixPointNumber & operator () (const double x);
35     /*inline*/ virtual FixPointNumber   operator +  ( ) const;
/*inline*/ virtual FixPointNumber   operator -  ( ) const;

/*inline*/ virtual FixPointNumber & operator ++ ( );
40     /*inline*/ virtual FixPointNumber & operator ++ (int);
/*inline*/ virtual FixPointNumber & operator -- ( );
/*inline*/ virtual FixPointNumber & operator -- (int);

/*inline*/ virtual FixPointNumber & operator += (const FixPointNumber& k);
/*inline*/ virtual FixPointNumber & operator -= (const FixPointNumber& k);
45     /*inline*/ virtual FixPointNumber & operator *= (const FixPointNumber& k);
/*inline*/ virtual FixPointNumber & operator /= (const FixPointNumber& k);

/*inline*/ virtual FixPointNumber & operator += (const double& k);
/*inline*/ virtual FixPointNumber & operator -= (const double& k);
50     /*inline*/ virtual FixPointNumber & operator *= (const double& k);
/*inline*/ virtual FixPointNumber & operator /= (const double& k);

```

389

```

5      /*inline*/ virtual FixPointNumber operator + (const FixPointNumber& k)
const;
6      /*inline*/ virtual FixPointNumber operator - (const FixPointNumber& k)
const;
7      /*inline*/ virtual FixPointNumber operator * (const FixPointNumber& k)
const;
8      /*inline*/ virtual FixPointNumber operator / (const FixPointNumber& k)
const;

9      /*inline*/ virtual FixPointNumber operator + (const double& k) const;
10     /*inline*/ virtual FixPointNumber operator - (const double& k) const;
11     /*inline*/ virtual FixPointNumber operator * (const double& k) const;
12     /*inline*/ virtual FixPointNumber operator / (const double& k) const;

13     /*inline*/ virtual int operator & (const int k ) const;

14     /*inline*/ virtual FixPointNumber operator << (const int k) const;
15     /*inline*/ virtual FixPointNumber operator >> (const int k) const;

16     /*inline*/ virtual char FractionBits() const;

protected:
17     FixPointData n;      // actual data
18     char fractionBits;
19     static char defaultFractionBits; // global default Precision
20     static char maxFractionBits;
21 }; // class FixPointNumber

22
23 #endif // _FixPointNumber_h_

:::::::::::
24 oop/FixPointNumber/Makefile
25 :::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
#      make           : compile only changed files and their depend
files
#      make all        : recompile all
#      make clean     : recompile all and remove unnecessary files
#
26
27 EXECUTABLE = go
28
29 SRC    = main.C Test.C Object.C FixPointNumber.C
30
31 RM = /bin/rm -fs
32
33 #
#      put -g here | to add debugging info to executable
#          V
34 CDEBUGFLAGS = -O -OPT:Olimit=2299

```

55

390

```

CCOPTIONS = -64

5 CC = CC

INCLUDES =
LIBS =
CFLAGS = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

10 OFILES = $(SRC:.C=.o)

$(EXECUTABLE): $(OFILES)
    $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

15 #
#      target: dependency \n tab rule
#
.C.o:
    $(CC) $(CFLAGS) -c $<

20 #
#      switch:: ; \n tab rule
#
clean: ;
    find . -name "*.*" -print -exec mv {} ~/dumpster \; ;
    find . -name "*~" -print -exec mv {} ~/dumpster \; ;
    find . -name "go" -print -exec mv {} ~/dumpster \; ;
    find . -name "core" -print -exec mv {} ~/dumpster \; ;

all: ;
    pmake -u

30 depend:
    makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

35 main.o: Test.h FixPointNumber.h
Test.o: Test.h FixPointNumber.h
Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h
FixPointNumber.o: FixPointNumber.h
::::::::::
40 oop/FixPointNumber/Test.C
::::::::::
// Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

#include "Test.h"
45 void Test::Run() { FixPointNumber::Demo(); }

// end of Test.cpp
::::::::::
50

```

55

391

```

oop/FixPointNumber/Test.h
:::::::::::
// Test.h          test class for OOP-classes
// (c) Ingmar Bitter '96

5      #ifndef _Test_h_           // prevent multiple includes
      #define _Test_h_

10     #include "FixPointNumber.h"

      class Test { public: static void Run(); };

      #endif // _Test_h_
:::::::::::
15     oop/Int/Global.h
:::::::::::
// Global.h        Global definitions
// (c) Ingmar Bitter '96

20     // Copyright, Mitsubishi Electric Information Technology Center
     // America, Inc., 1997, All rights reserved.

      #ifndef _Global_h_           // prevent multiple includes
      #define _Global_h_

25     #include <assert.h>
      #include <iostream.h> // ostream
      #include <typeinfo.h> // typeid(obj).name()
      #include <stdlib.h> // exit()

      //const unsigned char true   = 1
      //const unsigned char false  = 0
30

      #define Max(a,b) ((a) > (b)) ? (a) : (b)
      #define Min(a,b) ((a) < (b)) ? (a) : (b)
      #define ABS(a)    ((a) < 0 ) ? -(a) : (a)

35      //##define ERROR(str) { cerr << endl << (str) << " exiting ..." << endl; int
      k=999999, a[1]={1}; cout << a[k]; }
      #define ERROR(msg) { cerr << endl << "###Error### in file " << __FILE__ << " at "
      line " << __LINE__ << endl; cerr << (msg) << endl; cerr << "Exiting ..." <<
      endl; abort(); }

40      // avoid following warning
      // "/usr/include/CC/iostream.h", line 675: remark(1174):
      // variable "iostream_init" was declared but never referenced
      // } iostream_init ;
      #include <iostream.h>
      static void Dummy () { static void * dummy = &iostream_init; cout << dummy;
      Dummy(); }

45      #endif // _Global_h_
:::::::::::

```

50

55

```

oop/Int/Int.C
:::::::::::
5 // Int.cpp      class Int
// (c) Ingmar Bitter '96

#include "Int.h"

void Int::Demo()
10 {
    double a(2.2);
    Int k(3), m(k), n(k*k), p;
    cout << endl << "Demo of class " << typeid(k).name();
    cout << endl << "size : " << sizeof(Int) << " Bytes";
    cout << endl << "public members: - none";
15    cout << endl << "public member functions:";
    cout << endl << "    double a(2.2) = "<<a;
    cout << endl << "    Int k(3), m(k), n(k*k), p = "<<k<<, "<<m<<, "<<n<<, "<<p;
    cout << endl << "    p(-8) = "<<p(-8);
    cout << endl << "    k.Abs() = "<< k.Abs()<<"\tp.Abs() = "<< p.Abs();

20    cout << endl << "    k==k = "<<(k==k)<<"\tk<=k = "<<(k<=k)<<"\tk>k = "<<(k>k)
        <<"\tk>=9 = "<<(k==9)<<"\t9.<=k = "<<(9.<=k)<<"\tk>n =
    "<<(k>n);
    cout << endl << "    k!=k = "<<(k!=k)<<"\tk>=k = "<<(k>=k)<<"\tk>k = "<<(k>k)
        <<"\tk!=9 = "<<(k!=9)<<"\t9.>=k = "<<(9.>=k)<<"\tk>n =
25    "<<(k>n);

    cout << endl << "    ++k = "<<(++k) <<"\tm+n = "<<(m+n);
    cout     <<"\tm+a = "<<(m+a) <<"\tp+n = "<<(p+n);
    cout << endl << "    k++ = "<<(k++) <<"\tm-n = "<<(m-n);
    cout     <<"\tm-a = "<<(m-a) <<"\tp-n = "<<(p-n);
30    cout << endl << "    --k = "<<(--k) <<"\tm*n = "<<(m*n);
    cout     <<"\tm*a = "<<(m*a) <<"\ta*p = "<<(a*p);
    cout << endl << "    k-- = "<<(k--) <<"\tm/n = "<<(m/n);
    cout     <<"\tm/a = "<<(m/a)<<"\ta/p = "<<(a/p);

    Int array[5]; cout<< endl;
35    for (k(0); k<5; ++k)
        cout << "    array["<<k<<"] = "<<(array[k] = k);
    cout << endl << "End of demo of class " << typeid(k).name() << endl;
}      // Demo

40 //////////////////////////////////////////////////////////////////
// constructors & destructors

45 Int::Int      (const int   k) : n(k)      { }
Int::Int      (const Int & k) : n(k.n)      { }
Int::Int      (const Int * k) : n(k->n) { }

46 //////////////////////////////////////////////////////////////////
// casts to other data types

```

```

393
Int::operator int() const { return n; }

5   //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - class Object requirements

10  ostream & Int::Ostream (ostream & os) const { return os << n; }

15  //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - class Object requirements
double Int::Norm () const { return double(n); }

20  //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

Int Int::Abs () const { return Int((n<0) ? -n : n); }

25  Int & Int::operator () (const int k) { n=k; return *this; }

Int Int::operator +      () const { return Int(n); }
Int Int::operator -      () const { return Int(-n); }

30  Int Int::operator ++    ( ) { ++n; return *this; }
Int Int::operator ++    (int) { ++n; return *this; }
Int Int::operator --    ( ) { --n; return *this; }
Int Int::operator --    (int) { --n; return *this; }

35  Int & Int::operator += (const Int& k) { n+=k.n; return *this; }
Int & Int::operator -= (const Int& k) { n-=k.n; return *this; }
Int & Int::operator *= (const Int& k) { n*=k.n; return *this; }
Int & Int::operator /= (const Int& k) { n/=k.n; return *this; }

40  Int Int::operator +      (const Int& k) const { return Int(n+k.n); }
Int Int::operator -      (const Int& k) const { return Int(n-k.n); }
Int Int::operator *      (const Int& k) const { return Int(n*k.n); }
Int Int::operator /      (const Int& k) const { return Int(n/k.n); }

45  double Int::operator + (const double& x) const { return (n+x); }
double Int::operator - (const double& x) const { return (n-x); }
double Int::operator * (const double& x) const { return (n*x); }
double Int::operator / (const double& x) const { return (n/x); }

// define friend functions
double operator + (const double& x, const Int & k)      {      return x+k.n;
}

```

50

55

```

394
5      double operator - (const double& x, const Int& k) { return x-k.n;
      }
5      double operator * (const double& x, const Int& k) { return x*k.n;
      }
5      double operator / (const double& x, const Int& k) { return x/k.n;
      }

10     Int          operator + (const int n,           const Int& k) {
10     return n+k.n;   }
10     Int          operator - (const int n,           const Int& k) {
10     return n-k.n;   }
10     Int          operator * (const int n,           const Int& k) {
10     return n*k.n;   }
10     Int          operator / (const int n,           const Int& k) {
10     return n/k.n;   }

15     double & operator += (double& x, const Int& k) { . . . return x+=k.n; }
15     double & operator -= (double& x, const Int& k) { . . . return x-=k.n; }
15     double & operator *= (double& x, const Int& k) { . . . return x*=k.n; }
15     double & operator /= (double& x, const Int& k) { . . . return x/=k.n; }

20     int &          operator += (int n,           const Int& k) { . . .
20     return n+=k.n;   }
20     int &          operator -= (int n,           const Int& k) { . . .
20     return n-=k.n;   }
20     int &          operator *= (int n,           const Int& k) { . . .
20     return n*=k.n;   }
20     int &          operator /= (int n,           const Int& k) { . . .
20     return n/=k.n;   }

25

30 // end of Int

:::::::::::
cop/Int/Int.h
:::::::::::
35 // Int.h          class Int
// (c) Ingmar Bitter '96

#ifndef _Int_h_        // prevent multiple includes
#define _Int_h_

40 #include "Object.h"

45 class Int : virtual public Object {
public:
    static void Demo ();

    // constructors & destructors
    inline Int (const int k=0);
    inline Int (const Int & k);
    inline Int (const Int * k);

50
55

```

```

// casts to other data types
5   virtual inline operator int() const;

// show/set data & data properties
// - class Object requirements
    virtual ostream & Ostream (ostream &) const;

10  // - class CmpObj requirements
    virtual double Norm () const;

    // - local show/set functions
15  virtual inline Int Abs () const;

virtual inline Int & operator () (const int k); // set function
virtual inline Int operator + () const;
virtual inline Int operator - () const;
virtual inline Int operator ++ ();
20  virtual inline Int operator ++ (int);
virtual inline Int operator -- ();
virtual inline Int operator -- (int);
virtual inline Int & operator += (const Int& k);
virtual inline Int & operator -= (const Int& k);
25  virtual inline Int & operator *= (const Int& k);
virtual inline Int & operator /= (const Int& k);
    virtual inline Int operator + (const Int& k) const;
virtual inline Int operator - (const Int& k) const;
virtual inline Int operator * (const Int& k) const;
virtual inline Int operator / (const Int& k) const;
30  virtual inline double operator + (const double& x) const;
virtual inline double operator - (const double& x) const;
virtual inline double operator * (const double& x) const;
virtual inline double operator / (const double& x) const;

35  friend double operator + (const double& x, const Int& k);
friend double operator - (const double& x, const Int& k);
friend double operator * (const double& x, const Int& k);
friend double operator / (const double& x, const Int& k);
friend Int operator + (const int n, const Int& k);
40  friend Int operator - (const int n, const Int& k);
friend Int operator * (const int n, const Int& k);
friend Int operator / (const int n, const Int& k);
friend double & operator += (double& x, const Int& k);
friend double & operator -= (double& x, const Int& k);
45  friend double & operator *= (double& x, const Int& k);
friend double & operator /= (double& x, const Int& k);
friend int & operator += (int n, const Int& k);
friend int & operator -= (int n, const Int& k);
friend int & operator *= (int n, const Int& k);
friend int & operator /= (int n, const Int& k);

50  protected:
    int n;
}; // class INT

```

```

5      #endif // _Int_h_
::::::
oop/Int/Object.C
::::::
10     // Object.C    class Object    abstract base class for everything
     // (c) Ingmar Bitter '96

     // Copyright, Mitsubishi Electric Information Technology Center
     // America, Inc., 1997, All rights reserved.

15     #include "Object.h"

void Object::Demo ( )
{
    // A global default Demo-function
    //
    // All subclasses should implement their own Demo function
    // to demonstrate the classes abilities and syntax

    cout<<endl<<"Sorry, no Demo-function implemented"  <<endl;
    cout      <<"for this subclass of class Object :-(" <<endl;
25 } // Demo

///////////////////////////////
// show/set data & data properties
30
ostream & Object::Ostream (ostream & os) const
{
    return (os << typeid(*this).name() << "@" << (void *) this<<endl);
} // Ostream

35
ostream & operator << (ostream & os, const Object & obj)
{
    return obj.Ostream(os);
} // operator <<

40
ostream & operator << (ostream & os, const Object * obj)
{
    if (obj) return obj->Ostream(os);
    else      return os<<"NIL Pointer! ";
} // operator <<

45
const char * Object::BoolStr[] = {"Oh No ! :-( ,,:-)"};

// end of Object.C
::::::
50     oop/Int/Object.h
::::::

```

397

```

// Object.h      class Object          abstract base class for
everything
5 // (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

10 #ifndef _Object_h_                  // prevent multiple includes
#define _Object_h_

#include <string.h>    // strcmp
#include <iostream.h>
#include "Global.h"

15 class Object {
public:
    static void     Demo ( );
    // show/set data & data properties
20    virtual /*inline*/ ostream & Ostream ( ostream & os ) const;
    friend ostream & operator << ( ostream & os, const Object & obj );
    friend ostream & operator << ( ostream & os, const Object * obj );

public:
    static const char * BoolStr[];
25};

); // class Object

30 #endif // _Object_h_
:::oop/Int/Test.C
::::::::::
// Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

35 #include "Test.h"

void Test::Run() { Int::Demo(); }

40 // end of Test.cpp
:::oop/Int/Test.h
::::::::::
// Test.h        test class for OOP-classes
// (c) Ingmar Bitter '96

45 #ifndef _Test_h_                  // prevent multiple includes
#define _Test_h_

#include "Int.h"

50 class Test { public: static void Run(); };

```

55

```

5      #endif      // _Test__h_
:::oop/Int/main.C
::::::::::
// main.cpp
// (c) Ingmar Bitter '97

10     #include "Test.h"

main() { Test::Run(); } // main

// end of main.cpp
15     ::::oop/Int/main.h
::::::::::
// main.h

20     // Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997. All rights reserved.

::::::::::
oop/Matrix4x4/Makefile
::::::::::
25     #      Makefile for C++ programs (c) Ingmar Bitter '97
#
#      make          : compile only changed files and their depend
files
#      make all       : recompile all
#      make clean    : recompile all and remove unnecessary files
#
30

EXECUTABLE = go

SRC      = main.C Test.C Object.C Vector3D.C Matrix4x4.C
35
RM = /bin/rm -fs

#
#      put -g here | to add debugging info to executable
#          V
40     CDEBUGFLAGS = -O
CCOPTIONS   = -64

CC = CC

45     INCLUDES      =
LIBS        = -lm
CFLAGS      = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

OFILES =      $(SRC:.C=.o)
50     $(EXECUTABLE): $(OFILES)

```

```

399
$(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

5      #
#      target: dependency \n tab rule
#
.C.o:
      $(CC) $(CFLAGS) -c $<

10     #
#      switch:: ; \n tab rule
#
clean:    ;
      find . -name "*.o" -print -exec mv {} ~/dumpster \; ;
      find . -name "*~" -print -exec mv {} ~/dumpster \; ;
      find . -name "go" -print -exec mv {} ~/dumpster \; ;
      find . -name "core" -print -exec mv {} ~/dumpster \; ;

all:    ;
      pmake -u

20 depend:
      makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

25 main.o: Test.h Matrix4x4.h /usr/include/math.h /usr/include/sgidefs.h
main.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
main.o: /usr/include/assert.h Vector3D.h
Test.o: Test.h Matrix4x4.h /usr/include/math.h /usr/include/sgidefs.h
Test.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Test.o: /usr/include/assert.h Vector3D.h
30 Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h
Vector3D.o: Vector3D.h /usr/include/math.h /usr/include/sgidefs.h
Vector3D.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Vector3D.o: /usr/include/assert.h
35 Matrix4x4.o: Matrix4x4.h /usr/include/math.h /usr/include/sgidefs.h
Matrix4x4.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Matrix4x4.o: /usr/include/assert.h Vector3D.h
::::::::::
oop/Matrix4x4/Matrix4x4.C
::::::::::
40 // Matrix4x4.cpp
// (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

45 #include "Matrix4x4.h"

template <class _T_> void Matrix4x4<_T_>::Demo()
{
    Matrix4x4<_T_> a, b(1,0,0,0, 0,2,0,0, 0,0,3,0, 0,0,0,1);
50

```

```

400
cout << endl << "Demo of class " << typeid(a).name();
cout << endl << "size : " << sizeof(Matrix4x4<_T_>) << " Bytes";
cout << endl << "public member functions:";
cout << endl << " " << typeid(a).name() << " a, b(3,2,1); == " << a << ", " <<
5      &b;
/*
    cout << endl << " a<b;\t== " << (a<b) << "\t a<=b;\t== " <<
(a<=b)<<"\ta==b;\t== " << (a==b);
10     cout << endl << " a>b;\t== " << (a>b) << "\t a>=b;\t== " <<
(a>=b)<<"\ta!=b;\t== " << (a!=b);
*/
    cout << endl << " a=b;\t== " << (a==b);
    cout << endl << " +a;\t== " << (+a) ; // << "\t a+b;\t== " << (a+b);
15     cout << endl << " -a;\t== " << (-a) ; // << "\t a-b;\t== " << (a-b);
/*
    cout << endl << " a+=b;\t== " << (a+=b);
    cout << endl << " a-=b;\t== " << (a-=b);
    _T_ s; s = (_T_) 2.5;
    cout << endl << " " << typeid(s).name() << " s(2.5);\t== " << s;
20     cout << endl << " a*s;\t== " << (a*s) << "\t a*=s;\t== " << (a*=s);
*/
    Vector3D<_T_> v(2.5,2.5,2.5);
    cout << endl << " " << typeid(v).name() << " v(2.5,2.5,2.5);\t== " << v ;
    cout << endl << " a*v;\t== " << (a*v);
/*
25     cout << endl << " a.Determinant();\t== " << a.Determinant();
    cout << endl << " a.Det();\t== " << a.Det();
    cout << endl << " a.D();\t== " << a.D();
    cout << endl << " a.Transpose();\t== " << a.Transpose();
    cout << endl << " a.Trans();\t== " << a.Trans();
    cout << endl << " a.T();\t== " << a.T();
30     cout << endl << " a.Inverse();\t== " << a.Inverse();
    cout << endl << " a.Inv();\t== " << a.Inv();
    cout << endl << " a.I();\t== " << a.I();
*/
    cout << endl << "End of demo of class " << typeid(a).name() << endl;
35 } // Demo

////////////////////////////// // constructors & destructors //////////////////

40
template <class _T_> Matrix4x4<_T_>::Matrix4x4 ()
{
    element[0][0] = 1; element[1][0] = 0; element[2][0] = 0; element[3][0]
45    = 0;
    element[0][1] = 0; element[1][1] = 1; element[2][1] = 0; element[3][1]
    = 0;
    element[0][2] = 0; element[1][2] = 0; element[2][2] = 1; element[3][2]
    = 0;
}

```

50

55

```

401
    element[0][3] = 0; element[1][3] = 0; element[2][3] = 0; element[3][3]
= 1;
} // constructor

template <class _T_> Matrix4x4<_T_>::Matrix4x4 (const _T_ & ax, const _T_ & bx,
const _T_ & cx, const _T_ & dx,
10                                         const _T_ &
ay, const _T_ & by, const _T_ & cy, const _T_ & dy,
                                         const _T_ &
az, const _T_ & bz, const _T_ & cz, const _T_ & dz,
                                         const _T_ &
15                                         aw, const _T_ & bw, const _T_ & cw, const _T_ & dw)
{
    element[0][0] = ax; element[1][0] = bx; element[2][0] = cx;
element[3][0] = dx;
    element[0][1] = ay; element[1][1] = by; element[2][1] = cy;
20 element[3][1] = dy;
    element[0][2] = az; element[1][2] = bz; element[2][2] = cz;
element[3][2] = dz;
    element[0][3] = aw; element[1][3] = bw; element[2][3] = cw;
element[3][3] = dw;
} // constructor
25

template <class _T_> Matrix4x4<_T_>::Matrix4x4(const Matrix4x4<_T_> & matrix)
{
    register int m,k;

30    for (m=0; m<4; ++m)
        for (k=0; k<4; ++k)
            element[m][k] = matrix.element[m][k];
} // copy-constructor

35 ///////////////////////////////////////////////////////////////////
// casts to other data types

template <class _T_> Matrix4x4<_T_>::operator double() const { return Norm(); }

40 ///////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - class Object requirements

45 template <class _T_> ostream & Matrix4x4<_T_>::Ostream(ostream & os) const
{
    os <<" ";
    for (int m=0; m<4; ++m) {

```

50

55

402

```

5           os << "(";
for (int k=0; k<4; ++k) {
    os << element[m][k];
    if (k<3) os << ",";
}
os << ")^T";
if (m<3) os << ", ";
} os << " )";
return os;
} // Ostream

15 //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - class CmpObj requirements

20 template <class _T_> double Matrix4x4<_T_>::Norm() const
{
    return Determinant();
} // Norm

25 //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

30 template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator_()
(const _T_ & ax, const _T_ & bx, const _T_ & cx, const _T_ & dx,
 const _T_ & ay, const _T_ & by, const _T_ & cy, const _T_ & dy,
 const _T_ & az, const _T_ & bz, const _T_ & cz, const _T_ & dz,
 const _T_ & aw, const _T_ & bw, const _T_ & cw, const _T_ & dw)
35 {
    element[0][0] = ax;   element[1][0] = bx;   element[2][0] = cx;
element[3][0] = dx;
    element[0][1] = ay;   element[1][1] = by;   element[2][1] = cy;
element[3][1] = dy;
    element[0][2] = az;   element[1][2] = bz;   element[2][2] = cz;
element[3][2] = dz;
    element[0][3] = aw;   element[1][3] = bw;   element[2][3] = cw;
element[3][3] = dw;
    return *this;
} // operator ()
45

template <class _T_> Matrix4x4<_T_> Matrix4x4<_T_>::operator + () const {
    return Matrix4x4(*this);
}
template <class _T_> Matrix4x4<_T_> Matrix4x4<_T_>::operator - () const
{
50

```

55

```

        403
    return Matrix4x4( -element[0][0], -element[1][0], -element[2][0], -
5      element[3][0],
                           -element[0][1], -
element[1][1], -element[2][1], -element[3][1],
                           -element[0][2], -
element[1][2], -element[2][2], -element[3][2],
                           -element[0][3], -
element[1][3], -element[2][3], -element[3][3]);
10 // // operator +
}

template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::operator + (const
Matrix4x4<_T_> & m) const
{ cerr <<m<< "\n\n sorry, + not implemented \n\n"; return Matrix4x4();
15 } // operator +
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::operator - (const
Matrix4x4<_T_> & m) const
{ cerr <<m<< "\n\n sorry, - not implemented \n\n"; return Matrix4x4();
} // operator -
20 template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator +=(const
Matrix4x4<_T_> & m)
{ cerr <<m<< "\n\n sorry, += not implemented \n\n"; return *this;
} // operator +=
template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator --(const
Matrix4x4<_T_> & m)
25 { cerr <<m<< "\n\n sorry, -= not implemented \n\n"; return *this;
} // operator -=
/*
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::operator * (const _T_ & a)
const
{ cerr << "\n\n sorry, * not implemented \n\n"; return Matrix4x4();
} // operator *
*/
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::operator / (const _T_ & a)
const
{ cerr <<a<< "\n\n sorry, / not implemented \n\n"; return Matrix4x4();
} // operator /
35

template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator *=(const _T_ & a)
{ cerr <<a<< "\n\n sorry, *= not implemented \n\n"; return *this;
} // operator *=
template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator /=(const _T_ & a)
40 { cerr <<a<< "\n\n sorry, /= not implemented \n\n"; return *this;
} // operator /=
/*
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::operator * (const
Matrix4x4<_T_> & m) const
{ cerr <<m<< "\n\n sorry, * not implemented \n\n"; return Matrix4x4();
} // operator *
*/
template <class _T_> Matrix4x4<_T_> & Matrix4x4<_T_>::operator *=(const
Matrix4x4<_T_> & m)
45

```

50

55

```

404
{ cerr <<m<< "\n\n sorry, *= not           implemented \n\n"; return *this;
} // operator *=

5   template <class _T_> Vector3D<_T_>      Matrix4x4<_T_>::operator * (const
Vector3D<_T_> &v) const
{
    _T_ w(element[0][3]*v.X() + element[1][3]*v.Y() + element[2][3]*v.Z() +
element[3][3]);
10  return Vector3D<_T_>((element[0][0]*v.X() + element[1][0]*v.Y() +
element[2][0]*v.Z() + element[3][0])/w,
                           (element[0][1]*v.X() + element[1][1]*v.Y() + element[2][1]*v.Z() +
element[3][1])/w,
                           (element[0][2]*v.X() + element[1][2]*v.Y() + element[2][2]*v.Z() +
element[3][2])/w);
} // operator *

20  template <class _T_> Vector3D<double> Matrix4x4<_T_>::operator * (const
Vector3D<double> &v) const
{
    double w(element[0][3]*v.X() + element[1][3]*v.Y() + element[2][3]*v.Z() +
element[3][3]);
    return Vector3D<double>((element[0][0]*v.X() + element[1][0]*v.Y() +
element[2][0]*v.Z() + element[3][0])/w,
                           (element[0][1]*v.X() + element[1][1]*v.Y() + element[2][1]*v.Z() +
element[3][1])/w,
                           (element[0][2]*v.X() + element[1][2]*v.Y() + element[2][2]*v.Z() +
element[3][2])/w);
} // operator *

30  template <class _T_> Vector3D<FixPointNumber> Matrix4x4<_T_>::operator * (const
Vector3D<FixPointNumber> &v) const
{
    FixPointNumber w(element[0][3]*v.X() + element[1][3]*v.Y() +
element[2][3]*v.Z() + element[3][3]);
35  return Vector3D<FixPointNumber>((element[0][0]*v.X() + element[1][0]*v.Y() +
element[2][0]*v.Z() + element[3][0])/w,
                           (element[0][1]*v.X() + element[1][1]*v.Y() +
element[2][1]*v.Z() + element[3][1])/w,
                           (element[0][2]*v.X() + element[1][2]*v.Y() +
element[2][2]*v.Z() + element[3][2])/w);
} // operator *

40
45  template <class _T_> _T_ Matrix4x4<_T_>::Determinant( ) const { return D(); } //
Determinant
template <class _T_> _T_ Matrix4x4<_T_>::Det( )           const { return D(); } //
Det
template <class _T_> _T_ Matrix4x4<_T_>::D( )           const

```

50

55

```

405
( cerr << "\n\n sorry, Det() not           implemented \n\n"; return _T_();

5   } // D

template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::Transpose ( ) const {
return T(); } // Transpose
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::Trans      ( ) const {
return T(); } // Trans
10  template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::T          ( ) const {
{ cerr << "\n\n sorry, Trans() not implemented \n\n"; return Matrix4x4();
} // T

template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::Inverse ( ) const { return
T(); } // Inverse
15  template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::Inv       ( ) const { return
T(); } // Inv
template <class _T_> Matrix4x4<_T_>    Matrix4x4<_T_>::I          ( ) const {
{ cerr << "\n\n sorry, Inv() not implemented \n\n"; return Matrix4x4();
} // I

20

// end of Matrix4x4.cpp
:::::::::::
oop/Matrix4x4/Matrix4x4.h
:::::::::::
25  // Matrix4x4.h
// (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30  #ifndef _Matrix4x4_h_    // prevent multiple includes
#define _Matrix4x4_h_

35  #include <math.h>
#include "Object.h"
#include "Vector3D.h"
#include "FixPointNumber.h"

40

template <class _T_> class Matrix4x4 : public virtual Object {
public:
    static void        Demo      ( );
45
    // constructors & destructors
    /*inline*/ Matrix4x4 ();
    /*inline*/ Matrix4x4 (const _T_ & ax, const _T_ & bx, const _T_ & cx,
const _T_ & dx,
                           const _T_ & ay,
                           const _T_ & by, const _T_ & cy, const _T_ & dy,
                           const _T_ & az,
                           const _T_ & bz, const _T_ & cz, const _T_ & dz,
                           const _T_ & dz,
```

50

55

406

const _T_ & aw,

```

5      const _T_ & bw, const _T_ & cw, const _T_ & dw);
      /*inline*/ Matrix4x4 (const Matrix4x4<_T_> & matrix);

     // casts to other data types
     virtual /*inline*/ operator double() const;

10    // show/set data & data properties
     // - class Object requirements
     virtual          ostream &   Ostream (ostream & )   const;

     // - class CmpObj requirements
     virtual /*inline*/ double Norm ( ) const;

15    // - local show/set functions
     virtual /*inline*/ Matrix4x4<_T_> & operator () (const _T_ & ax, const _T_-
& bx, const _T_ & cx, const _T_ & dx,
                                         const _T_ & ay, const _T_-
& by, const _T_ & cy, const _T_ & dy,
                                         const _T_ & az, const _T_-
& bz, const _T_ & cz, const _T_ & dz,
                                         const _T_ & aw, const _T_-
& bw, const _T_ & cw, const _T_ & dw);

     virtual /*inline*/ Matrix4x4<_T_>   operator +  ( ) const;
     virtual /*inline*/ Matrix4x4<_T_>   operator -  ( ) const;

     virtual /*inline*/ Matrix4x4<_T_>   operator +  (const Matrix4x4<_T_> &)
30    const;
     virtual /*inline*/ Matrix4x4<_T_>   operator -  (const Matrix4x4<_T_> &)
const;
     virtual /*inline*/ Matrix4x4<_T_> & operator += (const Matrix4x4<_T_> &);
     virtual /*inline*/ Matrix4x4<_T_> & operator -= (const Matrix4x4<_T_> &);

35    // virtual /*inline*/ Matrix4x4<_T_>   operator *  (const _T_ &) const;
     virtual /*inline*/ Matrix4x4<_T_>   operator /  (const _T_ &) const;
     virtual /*inline*/ Matrix4x4<_T_> & operator *= (const _T_ &);
     virtual /*inline*/ Matrix4x4<_T_> & operator /= (const _T_ &);

40    // virtual      Matrix4x4<_T_>   operator *  (const Matrix4x4<_T_> &) const;
     virtual      Matrix4x4<_T_> & operator *= (const Matrix4x4<_T_> &);
     virtual      Vector3D<_T_>   operator *  (const Vector3D<_T_> &) const;
     virtual      Vector3D<double> operator *  (const Vector3D<double> &) const;
     virtual      Vector3D<FixPointNumber> operator *  (const
Vector3D<FixPointNumber> &) const;

45    virtual /*inline*/ _T_           Determinant ( ) const;
     virtual /*inline*/ _T_           Det            ( ) const;
     virtual      _T_             D              ( ) const;
     virtual /*inline*/ Matrix4x4<_T_> Transpose   ( ) const;

```

50

55

```

407
virtual /*inline*/ Matrix4x4<_T> Trans      ( ) const;
virtual      Matrix4x4<_T>   T      ( ) const;
virtual /*inline*/ Matrix4x4<_T> Inverse    ( ) const;
virtual /*inline*/ Matrix4x4<_T> Inv       ( ) const;
virtual      Matrix4x4<_T>   I      ( ) const;
protected:
    _T_ element[4][4];
}; // class Matrix4x4
10
#endif      // _Matrix4x4_h_
:::::::::::
oop/Matrix4x4/Test.C
:::::::::::
// Test.cpp      testclass for OOP-classes
15
// (c) Ingmar Bitter '96

#include "Test.h"

void Test::Run() { Matrix4x4<int>::Demo(); }

20
// end of Test.cpp
:::::::::::
oop/Matrix4x4/Test.h
:::::::::::
// Test.h          test class for OOP-classes
// (c) Ingmar Bitter '96
25
#ifndef _Test_h_           // prevent multiple includes
#define _Test_h_

#include "Matrix4x4.h"

30
class Test { public: static void Run(); };

#endif      // _Test_h_
:::::::::::
oop/ModInt/Int.C
:::::::::::
35
// Int.cpp        class Int
// (c) Ingmar Bitter '96

#include "Int.h"

40
void Int::Demo()
{
    double a(2.2);
    Int k(3), m(k), n(k*k), p;
    cout << endl << "Demo of class " << typeid(k).name();
    cout << endl << "size : " << sizeof(Int) << " Bytes";
45
    cout << endl << "public members: - none";
    cout << endl << "public member functions:";
    cout << endl << "    double a(2.2) = "<<a;
    cout << endl << "    Int k(3), m(k), n(k*k), p = "<<k<<, "<<m<<, "<<n<<, "<<p;

```

50

55

```

408
cout << endl << " p(-8) = "<<p(-     8);
cout << endl << " k.Abs() = "<< k.Abs()<<"\tp.Abs() = "<< p.Abs();
5
cout << endl << " k==k = "<<(k==k)<<"\tk<=k = "<<(k<=k)<<"\tk<k = "<<(k<k)
<<"\tk>=9 = "<<(k==9)<<"\t9.<=k = "<<(9.<=k)<<"\tk<n =
"<<(k<n);
cout << endl << " k!=k = "<<(k!=k)<<"\tk>=k = "<<(k>=k)<<"\tk>k = "<<(k>k)
<<"\tk!=9 = "<<(k!=9)<<"\t9.>=k = "<<(9.>=k)<<"\tk>n =
10   "<<(k>n);

cout << endl << " ++k = "<<(++k) <<"\tm+n = "<<(m+n);
cout           <<"\tm+a = "<<(m+a) <<"\tp+=n = "<<(p+=n);
cout << endl << " k++ = "<<(k++) <<"\tm-n = "<<(m-n);
cout           <<"\tm-a = "<<(m-a) <<"\tp-=n = "<<(p-=n);
15
cout << endl << " --k = "<<(--k) <<"\tm*n = "<<(m*n);
cout           <<"\tm*a = "<<(m*a) <<"\ta*=p = "<<(a*=p);
cout << endl << " k-- = "<<(k--) <<"\tm/n = "<<(m/n);
cout           <<"\tm/a = "<<(m/a)<<"\ta/=p = "<<(a/=p);

20
Int array[5]; cout<< endl;
for (k(0); k<5; ++k)
    cout << " array["<<k<<"] = "<<(array[k] = k);
cout << endl <<"End of demo of class " << typeid(k).name() << endl;
} // Demo
25

////////////////////////////// // constructors & destructors

30
Int::Int      (const int   k) : n(k)      { }
Int::Int      (const Int & k) : n(k.n)      { }
Int::Int      (const Int * k) : n(k->n) { }

////////////////////////////// // casts to other data types

35
Int::operator int() const { return n; }

////////////////////////////// // show/set data & data properties
// - class Object requirements

40
ostream &   Int::Ostream (ostream & os) const { return os << n; }

////////////////////////////// // show/set data & data properties
// - class Object requirements

45
double Int::Norm () const { return double(n); }

50

```

```

////////// show/set data & data properties
5 // - local show/set functions

Int Int::Abs () const { return Int((n<0) ? -n : n); }

10 Int & Int::operator () (const int k) { n=k; return *this; }

Int Int::operator + ( ) const { return Int(n); }
Int Int::operator - ( ) const { return Int(-n); }

15 Int Int::operator ++ ( ) { ++n; return *this; }
Int Int::operator ++ (int) { ++n; return *this; }
Int Int::operator -- ( ) { --n; return *this; }
Int Int::operator -- (int) { --n; return *this; }

20 Int Int::operator += (const Int& k) { n+=k.n; return *this; }
Int Int::operator -= (const Int& k) { n-=k.n; return *this; }
Int Int::operator *= (const Int& k) { n*=k.n; return *this; }
Int Int::operator /= (const Int& k) { n/=k.n; return *this; }

25 Int Int::operator + (const Int& k) const { return Int(n+k.n); }
Int Int::operator - (const Int& k) const { return Int(n-k.n); }
Int Int::operator * (const Int& k) const { return Int(n*k.n); }
Int Int::operator / (const Int& k) const { return Int(n/k.n); }

30 double Int::operator + (const double& x) const { return (n+x); }
double Int::operator - (const double& x) const { return (n-x); }
double Int::operator * (const double& x) const { return (n*x); }
double Int::operator / (const double& x) const { return (n/x); }

35 // define friend functions
double operator + (const double& x, const Int& k) { return x+k.n; }
double operator - (const double& x, const Int& k) { return x-k.n; }
double operator * (const double& x, const Int& k) { return x*k.n; }
double operator / (const double& x, const Int& k) { return x/k.n; }

40 Int operator + (const int n, const Int& k) {
    return n+k.n;
}
Int operator - (const int n, const Int& k) {
    return n-k.n;
}
Int operator * (const int n, const Int& k) {
    return n*k.n;
}
45 Int operator / (const int n, const Int& k) {
    return n/k.n;
}

double & operator += (double& x, const Int& k) { return x+=k.n; }

50

```

```

410
double & operator -= (double& x, const Int& k) { return x-=k.n;
}
5 double & operator *= (double& x, const Int& k) { return x*=k.n; }
double & operator /= (double& x, const Int& k) { return x/=k.n; }

int & operator += (int n, const Int& k) {
    return n+=k.n; }
int & operator -= (int n, const Int& k) { return
10   n-=k.n; }
int & operator *= (int n, const Int& k) { return
n*=k.n; }
int & operator /= (int n, const Int& k) { return
n/=k.n; }

15 // end of Int

:::::::::::
oop/ModInt/Int.h
:::::::::::
20 // Int.h      class Int
// (c) Ingmar Bitter '96

#ifndef _Int_h_           // prevent multiple includes
#define _Int_h_
25 #include "Object.h"
#include <typeinfo.h>

30 class Int : virtual public Object {
public:
    static void Demo ();

    // constructors & destructors
    inline Int (const int k=0);
    inline Int (const Int & k);
35    inline Int (const Int * k);

    // casts to other data types
    virtual inline operator int() const;

    // show/set data & data properties
    // - class Object requirements
40    virtual ostream & Ostream (ostream &) const;

    // - class CmpObj requirements
    virtual double Norm () const;

45    // - local show/set functions
    virtual inline Int Abs () const;

    virtual inline Int & operator () (const int k); // set function
}

50

```

```

5           virtual inline Int      operator + ( ) const;
           virtual inline Int      operator - ( ) const;
           virtual inline Int      operator ++ ( );
           virtual inline Int      operator ++ (int);
           virtual inline Int      operator -- ( );
           virtual inline Int      operator -- (int);
           virtual inline Int      operator += (const Int& k);
           virtual inline Int      operator -= (const Int& k);
           virtual inline Int      operator *= (const Int& k);
           virtual inline Int      operator /= (const Int& k);
           virtual inline Int      operator + (const Int& k) const;
10          virtual inline Int      operator - (const Int& k) const;
           virtual inline Int      operator * (const Int& k) const;
           virtual inline Int      operator / (const Int& k) const;
           virtual inline double    operator + (const double& x) const;
           virtual inline double    operator - (const double& x) const;
           virtual inline double    operator * (const double& x) const;
           virtual inline double    operator / (const double& x) const;
20
25          friend double   operator + (const double& x, const Int& k);
          friend double   operator - (const double& x, const Int& k);
          friend double   operator * (const double& x, const Int& k);
          friend double   operator / (const double& x, const Int& k);
          friend Int      operator + (const int n, const Int& k);
          friend Int      operator - (const int n, const Int& k);
          friend Int      operator * (const int n, const Int& k);
          friend Int      operator / (const int n, const Int& k);
          friend double & operator += (double& x, const Int& k);
30          friend double & operator -= (double& x, const Int& k);
          friend double & operator *= (double& x, const Int& k);
          friend double & operator /= (double& x, const Int& k);
          friend int &     operator += (int n, const Int& k);
          friend int &     operator -= (int n, const Int& k);
          friend int &     operator *= (int n, const Int& k);
          friend int &     operator /= (int n, const Int& k);
35
protected:
  int n;
}; // class INT

40 #endif // _Int_h_

:::::::::::
oop/ModInt/Makefile
45 :::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
#      make      : compile only changed files and their depend files
#      make all   : recompile all
#      make clean : recompile all and remove unnecessary files
50 #
EXECUTABLE = go

```

```

SRC    = main.C Test.C Object.C ModInt.C
5
RM = /bin/rm -fs

#
#      put -g here | to add debugging info to executable
#                           V
10 CDEBUGFLAGS = -O
#CCOPTIONS = -64
CCOPTIONS     = -n32 -mips4 -r10000

CC = CC
15
INCLUDES    =
LIBS        =
CFLAGS       = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

OFILES =    $(SRC:.C=.o)
20
$(EXECUTABLE): $(OFILES)
               $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

#
25 #      target: dependency \n tab rule
#
.C.o:
               $(CC) $(CFLAGS) -c $<

#
30 #      switch:: ; \n tab rule
#
clean:      ;
               find . -name "*.*" -print -exec rm {} \; ;
               find . -name "*~" -print -exec rm {} \; ;
               find . -name "go" -print -exec rm {} \; ;
               find . -name "core" -print -exec rm {} \; ;

all:      ;
               pmake -u
40 depend:
               makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

45 main.o: Test.h ModInt.h Object.h /usr/include/string.h
main.o: /usr/include/standards.h Global.h /usr/include/assert.h
main.o: /usr/include/stdlib.h /usr/include/sgidefs.h /usr/include/limits.h
Test.o: Test.h ModInt.h Object.h /usr/include/string.h
Test.o: /usr/include/standards.h Global.h /usr/include/assert.h
Test.o: /usr/include/stdlib.h /usr/include/sgidefs.h /usr/include/limits.h
50 Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h

```

```

413
Object.o: /usr/include/assert.h           /usr/include/stdlib.h
/usr/include/sgidefs.h
5      ModInt.o: ModInt.h Object.h /usr/include/string.h /usr/include/standards.h
ModInt.o: Global.h /usr/include/assert.h /usr/include/stdlib.h
ModInt.o: /usr/include/sgidefs.h /usr/include/limits.h
::::::
oop/ModInt/ModInt.C
::::::
10    // ModInt.C class ModInt
// (c) Hanspeter Pfister '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15    #include "ModInt.h"

void ModInt::Demo()
{
    ModInt k(3), l(5,3), m(1), n;

20    cout << endl << "Demo of class " << typeid(k).name();
    cout << endl << "size : " << sizeof(ModInt) << " Bytes";
    cout << endl << "public members: - none";
    cout << endl << "public member functions:";

25    cout << endl << "Constructors";
    cout << endl << " ModInt k(3), l(5,3), m(1), n = "<<k<<, "<<l<<, "<<m<<",
"<<n;
        cout << endl << Object::BoolStr[k==3 && l==2 && m==2 && n==0];

30    cout << endl << " k(0,3), k(1,3), k(2,3), k(3,3), k(4,3) = "
        <<k(0,3)<<, "<<k(1,3)<<, "<<k(2,3)<<, "<<k(3,3)<<, "<<k(4,3);
    cout << endl << Object::BoolStr[k(0,3)==0 && k(1,3)==1 && k(2,3)==2 &&
k(3,3)==0 && k(4,3)==1];

35    cout << endl << " k(0,4), k(1,4), k(2,4), k(3,4), k(4,4) = "
        <<k(0,4)<<, "<<k(1,4)<<, "<<k(2,4)<<, "<<k(3,4)<<, "<<k(4,4);
    cout << endl << Object::BoolStr[k(0,4)==0 && k(1,4)==1 && k(2,4)==2 &&
k(3,4)==3 && k(4,4)==0];

40    cout << endl << " k(-1,3), k(-2,3), k(-3,3), k(-4,3) = "
        <<k(-1,3)<<, "<<k(-2,3)<<, "<<k(-3,3)<<, "<<k(-4,3);
    cout << endl << Object::BoolStr[k(-1,3)==2 && k(-2,3)==1 && k(-3,3)==0 &&
k(-4,3)==2];

45    cout << endl << " k.Abs() = "<< k.Abs();
    cout << endl << Object::BoolStr[k.Abs()==2];

cout << endl << "Comparisons";

cout << endl << " k==k = "<<(k==k)<<"\tk<=k = "<<(k<=k)<<"\tk>k = "<<(k>k)
<<"\tk==9 = "<<(k==9)<<"\tk9.<=k = "<<(9.<=k)<<"\tk<n = "<<(k<n);

50

```

```

414
      cout << endl <<
      && (k<k) ==0 && (k==9)==0
5          Object::BoolStr[(k==k)==1 && (k<k)==1
      && (9.<=k)==0 && (k<n)==0];

      cout << endl <<"  k!=k = "<<(k!=k)<<"\tk>k = "<<(k>k)<<"\tk>k = "<<(k>k)
      <<"\tk!=9 = "<<(k!=9)<<"\tk9.>k = "<<(9.>k)<<"\tk>n = "<<(k>n);
      cout << endl << Object::BoolStr[(k!=k)==0 && (k>k)==1 && (k>k)==0 &&
(k!=9)==1
10             && (9.>k)==1 && (k>n)==1];

      cout << endl <<"Unary Operators";

      cout << endl <<"  ++k = "<<(++k);
      cout << endl << Object::BoolStr[k==0];
15      cout << endl <<"  k++ = "<<(k++);
      cout << endl << Object::BoolStr[k==1];
      cout << endl <<"  --k = "<<(--k);
      cout << endl << Object::BoolStr[k==0];
      cout << endl <<"  k-- = "<<(k--);
20      cout << endl << Object::BoolStr[k==2];
      cout << endl <<"  k-- = "<<(k--);
      cout << endl << Object::BoolStr[k==1];
      cout << endl <<"  k-- = "<<(k--);
      cout << endl << Object::BoolStr[k==0];
25      cout << endl <<"  k-- = "<<(k--);
      cout << endl << Object::BoolStr[k==2];
      cout << endl <<"  k-- = "<<(k--);
      cout << endl << Object::BoolStr[k==1];

      cout << endl <<"ModInt op ModInt";
      k(1,5);
30      n(2,5);
      cout << endl <<"  k = " << k << "\tn = " << n;

      cout << endl <<"  k+=n = "<<(k+=n);
      cout << endl << Object::BoolStr[k==3];
      cout << endl <<"  k*=n = "<<(k*=n);
35      cout << endl << Object::BoolStr[k==1];
      cout << endl <<"  k-=n = "<<(k-=n);
      cout << endl << Object::BoolStr[k==4];
      cout << endl <<"  k/=n = "<<(k/=n);
      cout << endl << Object::BoolStr[k==2];

40      k(4,5);
      n(2,5);
      cout << endl <<"  k = " << k << "\tn = " << n;

      cout << endl <<"  k+n = "<<(k+n) <<"\tk-n = "<<(k-n);
      cout << endl <<"\tk*n = "<<(k*n) <<"\tk/n = "<<(k/n);
45      cout << endl << Object::BoolStr[(k+n)==1 && (k-n)==2 && (k*n)==3 &&
(k/n)==2];

      cout << endl <<"ModInt op double";

```

50

55

415

```

k(4,5);
double a(2.2);
cout << endl << " k = " << k << "\ta = " << a;

cout << endl << " k+=a = "<<(k+=a);
cout << endl << Object::BoolStr[k==1];
cout << endl << " k-=a = "<<(k-=a);
cout << endl << Object::BoolStr[k==4];
cout << endl << " k*=a = "<<(k*=a);
cout << endl << Object::BoolStr[k==3];
cout << endl << " k/=a = "<<(k/=a);
cout << endl << Object::BoolStr[k==1];

10   cout << endl << " k+a = "<<(k+a) << "\tk-a = "<<(k-a);
      cout << endl << " \tk*a = "<<(k*a) << "\tk/a = "<<(k/a);
      cout << endl << Object::BoolStr[(k+a)==3 && (k-a)==4 &&
                                  (k*a)==2 && (k/a)==0];

15   cout << endl << "double op ModInt";
k(4,5);
cout << endl << " k = " << k << "\ta = " << a;

cout << endl << " a+k = "<<(a+k) << "\ta-k = "<<(a-k);
cout << endl << " \ta*k = "<<(a*k) << "\ta/k = "<<(a/k);
20   cout << endl << Object::BoolStr[(a+k==(2.2+4) && (a-k==(2.2-4) &&
                                  (a*k==(2.2*4) && (a/k==(2.2/4))];

cout << endl << " a+=k = "<<(a+=k);
cout << endl << Object::BoolStr[a==6.2];
25   cout << endl << " a-=k = "<<(a-=k);
      cout << endl << Object::BoolStr[a==2.2];
      cout << endl << " a*=k = "<<(a*=k);
      cout << endl << Object::BoolStr[a==8.8];
      cout << endl << " a/=k = "<<(a/=k);
      cout << endl << Object::BoolStr[a==2.2];

30   ModInt array[6];
int i = 0;
      for (k(0); i<10; ++k, ++i)
        cout << endl << " array["<<k<<"] = "<<(array[k] = k);

35   cout << endl << "End of demo of class " << typeid(k).name() << endl;
} // Demo

40   /////////////////////////////////
45   // MyMod: Calculate modulus for neg numbers, too.

int ModInt::MyMod (int n, unsigned int m)
{
50   while ( n < 0 ) {
      n += m;
    }
    return ( n % m );
}

```

55

416

```

}
5 //////////////////////////////////////////////////////////////////
// constructors & destructors

ModInt::ModInt (const int k, const unsigned int modulus)
    : n(MyMod(k, modulus)), m(modulus) { }
ModInt::ModInt (const ModInt & k)
    : n(MyMod(k.n, k.m)), m(k.m) { }
ModInt::ModInt (const ModInt * k)
    : n(MyMod(k->n, k->m)), m(k->m) { }

10 //////////////////////////////////////////////////////////////////
// casts to other data types

15 ModInt::operator int() const { return n; }

19 //////////////////////////////////////////////////////////////////
// show/set data & data properties
20 //
// - class Object requirements

25 ostream & ModInt::Ostream (ostream & os) const { return (os << n << ":" << m); }

29 //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - class Object requirements

33 double ModInt::Norm () const { return double(n); }

37 //////////////////////////////////////////////////////////////////
// show/set data & data properties
//
// - local show/set functions

38 ModInt ModInt::Abs () const { return ModInt((n<0) ? -n : n); }

42 ModInt & ModInt::operator () (const int k, const unsigned int modulus)
{
    n = MyMod(k, modulus);
    m = modulus;
    return *this;
}

46 ModInt & ModInt::operator () (const int k)
{
    n = MyMod(k, m);
    return *this;
}

50 //////////////////////////////////////////////////////////////////

```

55

```

// operators
//
5   // Unary operators
ModInt ModInt::operator +      ( ) const { return ModInt(n); }
ModInt ModInt::operator -      ( ) const { return ModInt(-n); }
ModInt & ModInt::operator ++     ( ) { n = ++n % m; return *this; }
ModInt & ModInt::operator ++     (int) { n = ++n % m; return *this; }
10  ModInt & ModInt::operator --     ( ) { n = MyMod(--n, m); return *this; }
ModInt & ModInt::operator --     (int) { n = MyMod(--n, m); return *this; }

// ModInt op ModInt
ModInt & ModInt::operator +=    (const ModInt& k) { n = MyMod((n+k.n), m); return
15  *this; }
ModInt & ModInt::operator -=    (const ModInt& k) { n = MyMod((n-k.n), m); return
*this; }
ModInt & ModInt::operator *=    (const ModInt& k) { n = MyMod((n*k.n), m); return
*this; }
ModInt & ModInt::operator /=    (const ModInt& k) { n = MyMod((n/k.n), m); return
20  *this; }
ModInt ModInt::operator +      (const ModInt& k) const { return ModInt(n+k.n, m); }
ModInt ModInt::operator -      (const ModInt& k) const { return ModInt(n-k.n, m); }
ModInt ModInt::operator *      (const ModInt& k) const { return ModInt(n*k.n, m); }
25  ModInt ModInt::operator /      (const ModInt& k) const { return ModInt(n/k.n, m); }

// ModInt op double
ModInt & ModInt::operator +=    (const double& x) { n = MyMod((n+x), m); return
30  *this; }
ModInt & ModInt::operator -=    (const double& x) { n = MyMod((n-x), m); return
*this; }
ModInt & ModInt::operator *=    (const double& x) { n = MyMod((n*x), m); return
*this; }
ModInt & ModInt::operator /=    (const double& x) { n = MyMod((n/x), m); return
35  *this; }
ModInt ModInt::operator +      (const double& x) const { return ModInt(n+x, m); }
ModInt ModInt::operator -      (const double& x) const { return ModInt(n-x, m); }
ModInt ModInt::operator *      (const double& x) const { return ModInt(n*x, m); }
ModInt ModInt::operator /      (const double& x) const { return ModInt(n/x, m); }
40

// double op ModInt
double & operator +=    (double& x, const ModInt& k) { return x+=k.n; }
double & operator -=    (double& x, const ModInt& k) { return x-=k.n; }
double & operator *=    (double& x, const ModInt& k) { return x*=k.n; }
double & operator /=    (double& x, const ModInt& k) { return x/=k.n; }
45  double operator +    (const double& x, const ModInt& k) { return x+k.n;
}
double operator -    (const double& x, const ModInt& k) { return x-k.n;
}

```

```

418
double      operator *  (const double&  x, const ModInt& k)      {      return
x*k.n;
}
double      operator /  (const double& x, const ModInt& k)  {      return x/k.n;
}

5

// ModInt op int
ModInt & ModInt::operator =      (const int& i) { n = MyMod(i, m); return
*this; }
ModInt & ModInt::operator +=     (const int& i) { n = MyMod((n+i), m); return
*this; }
ModInt & ModInt::operator -=     (const int& i) { n = MyMod((n-i), m); return
*this; }
ModInt & ModInt::operator *=     (const int& i) { n = MyMod((n*i), m); return
*this; }
ModInt & ModInt::operator /=     (const int& i) { n = MyMod((n/i), m); return
*this; }
ModInt ModInt::operator +      (const int& i) const { return ModInt(n+i, m); }
ModInt ModInt::operator -      (const int& i) const { return ModInt(n-i, m); }
ModInt ModInt::operator *      (const int& i) const { return ModInt(n*i, m); }
ModInt ModInt::operator /      (const int& i) const { return ModInt(n/i, m); }

10

20

// int op ModInt
int & operator += (int& i, const ModInt& k)      {      return i+=k.n; }
int & operator -= (int& i, const ModInt& k)      {      return i-=k.n; }
int & operator *= (int& i, const ModInt& k)      {      return i*=k.n; }
int & operator /= (int& i, const ModInt& k)      {      return i/=k.n; }
int   operator + (const int& i, const ModInt& k) {      return i+k.n;   }
int   operator - (const int& i, const ModInt& k) {      return i-k.n;   }
int   operator * (const int& i, const ModInt& k) {      return i*k.n;   }
int   operator / (const int& i, const ModInt& k) {      return i/k.n;   }

25

30

35

40

// ModInt op long
ModInt & ModInt::operator += (const long& i) { n = MyMod((n+i), m); return
*this; }
ModInt & ModInt::operator -= (const long& i) { n = MyMod((n-i), m); return
*this; }
ModInt & ModInt::operator *= (const long& i) { n = MyMod((n*i), m); return
*this; }
ModInt & ModInt::operator /= (const long& i) { n = MyMod((n/i), m); return
*this; }
ModInt ModInt::operator +      (const long& i) const { return ModInt(n+i, m); }
ModInt ModInt::operator -      (const long& i) const { return ModInt(n-i, m); }
ModInt ModInt::operator *      (const long& i) const { return ModInt(n*i, m); }
ModInt ModInt::operator /      (const long& i) const { return ModInt(n/i, m); }

45

// long op ModInt
long & operator += (long& i, const ModInt& k) {      return i+=k.n; }
long & operator -= (long& i, const ModInt& k) {      return i-=k.n; }
long & operator *= (long& i, const ModInt& k) {      return i*=k.n; }
long & operator /= (long& i, const ModInt& k) {      return i/=k.n; }
long   operator + (const long& i, const ModInt& k) {      return i+k.n;   }
long   operator - (const long& i, const ModInt& k) {      return i-k.n;   }
long   operator * (const long& i, const ModInt& k) {      return i*k.n;   }

50

```

```

        419
long operator / (const long& i, const ModInt& k) {      return i/k.n;      }

5 // end of ModInt

::::::::
oop/ModInt/ModInt.h
:::::::
// ModInt.h      class ModInt
10 // (c) Hanspeter Pfister '97

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

15 #ifndef _ModInt_h_           // prevent multiple includes
#define _ModInt_h_

#include "Object.h"
#include <iostream.h>
#include <typeinfo.h>
20 #include <limits.h>

class ModInt : virtual public Object {
public:
    static void Demo ();
25    // constructors & destructors
    ModInt (const int k=0, const unsigned int modulus=INT_MAX);
    ModInt (const ModInt & k);
    ModInt (const ModInt * k);

30    // casts to other data types
    virtual operator int() const;

    // show/set data & data properties
    // - class Object requirements
35    virtual ostream & Ostream (ostream & os) const;
    // - class CmpObj requirements
    virtual double Norm () const;

    // - local show/set functions
40    virtual ModInt Abs () const;

    virtual ModInt & operator () (const int k); // set function
    virtual ModInt & operator () (const int k, const unsigned int modulus);

45    // Unary operators
    virtual ModInt operator + () const;
    virtual ModInt operator - () const;
    virtual ModInt & operator ++ ();
    virtual ModInt & operator ++ (int);
    virtual ModInt & operator -- ();

```

50

55

420

```

virtual ModInt & operator -- (int);

5   // ModInt op ModInt
virtual ModInt & operator += (const ModInt& k);
virtual ModInt & operator -= (const ModInt& k);
virtual ModInt & operator *= (const ModInt& k);
virtual ModInt & operator /= (const ModInt& k);
10  virtual ModInt operator + (const ModInt& k) const;
virtual ModInt operator - (const ModInt& k) const;
virtual ModInt operator * (const ModInt& k) const;
virtual ModInt operator / (const ModInt& k) const;

15  // ModInt op double
virtual ModInt & operator += (const double& x);
virtual ModInt & operator -= (const double& x);
virtual ModInt & operator *= (const double& x);
virtual ModInt & operator /= (const double& x);
20  virtual ModInt operator + (const double& x) const;
virtual ModInt operator - (const double& x) const;
virtual ModInt operator * (const double& x) const;
virtual ModInt operator / (const double& x) const;

25  // double op ModInt
friend double & operator += (double& x, const ModInt& k);
friend double & operator -= (double& x, const ModInt& k);
friend double & operator *= (double& x, const ModInt& k);
friend double & operator /= (double& x, const ModInt& k);
30  friend double operator + (const double& x, const ModInt& k);
friend double operator - (const double& x, const ModInt& k);
friend double operator * (const double& x, const ModInt& k);
friend double operator / (const double& x, const ModInt& k);

35  // ModInt op int
virtual ModInt & operator = (const int& i);
virtual ModInt & operator += (const int& i);
virtual ModInt & operator -= (const int& i);
virtual ModInt & operator *= (const int& i);
virtual ModInt & operator /= (const int& i);
40  virtual ModInt operator + (const int& i) const;
virtual ModInt operator - (const int& i) const;
virtual ModInt operator * (const int& i) const;
virtual ModInt operator / (const int& i) const;

45  // int op ModInt
friend int & operator += (int& i, const ModInt& k);
friend int & operator -= (int& i, const ModInt& k);
friend int & operator *= (int& i, const ModInt& k);
friend int & operator /= (int& i, const ModInt& k);
50  friend int operator + (const int& i, const ModInt& k);
friend int operator - (const int& i, const ModInt& k);
friend int operator * (const int& i, const ModInt& k);
friend int operator / (const int& i, const ModInt& k);

```

```

        421
5   // ModInt op long
virtual ModInt & operator += (const long& i);
virtual ModInt & operator -= (const long& i);
virtual ModInt & operator *= (const long& i);
virtual ModInt & operator /= (const long& i);
virtual ModInt operator + (const long& i) const;
virtual ModInt operator - (const long& i) const;
10  virtual ModInt operator * (const long& i) const;
virtual ModInt operator / (const long& i) const;

15  // long op ModInt
friend long & operator += (long& i, const ModInt& k);
friend long & operator -= (long& i, const ModInt& k);
friend long & operator *= (long& i, const ModInt& k);
friend long & operator /= (long& i, const ModInt& k);
friend long operator + (const long& i, const ModInt& k);
friend long operator - (const long& i, const ModInt& k);
friend long operator * (const long& i, const ModInt& k);
20  friend long operator / (const long& i, const ModInt& k);

protected:
    int n;
    unsigned int m;

25  private:
    int MyMod(int n, unsigned int m);

}; // class MODINT

30  #endif      // _ModInt_h_

:::::::::::
oop/ModInt/Test.C
:::::::::::
35  // Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

#include "Test.h"

40  void Test::Run() { ModInt::Demo(); }

// end of Test.cpp
:::::::::::
oop/ModInt/Test.h
45  :::::::::::
// Test.h          test class for OOP-classes
// (c) Ingmar Bitter '96

#ifndef _Test_h_           // prevent multiple includes
50  #define _Test_h_

#include "ModInt.h"

```

422

```

class Test { public: static void Run(); };

5      #endif      // _Test_h_
:::::::::::
oop/Timer/Makefile
:::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
#      make                  : compile only changed files and their depend
10     files
#      make all              : recompile all
#      make clean             : recompile all and remove unnecessary files
#
15
EXECUTABLE = go

SRC    = main.C Test.C Object.C Timer.C

20     RM = /bin/rm -fs

#
#      put -g here | to add debugging info to executable
#          v
CDEBUGFLAGS = -g
25     COPTIONS   = -64

CC = CC

INCLUDES =
LIBS =
30     CFLAGS      = $(COPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

OFILES = $(SRC:.C=.o)

$(EXECUTABLE): $(OFILES)
35           $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

#
#      target: dependency \n tab rule
#
40     .C.o:
           $(CC) $(CFLAGS) -c $<

#
#      switch:: ; \n tab rule
#
45     clean: ;
           find . -name "*.*" -print -exec mv {} ~/dumpster \; ;
           find . -name "*~"  -print -exec mv {} ~/dumpster \; ;
           find . -name "go"   -print -exec mv {} ~/dumpster \; ;
           find . -name "core" -print -exec mv {} ~/dumpster \; ;

```

50

55

```

        423
all: ;
      pmake -u
5
depend:
      makedepend -- $(CFLAGS) -- $(SRC)

# DO NOT DELETE

10 main.o: Test.h Timer.h /usr/include/sys/time.h /usr/include/standards.h
main.o: /usr/include/sgidefs.h /usr/include/sys/times.h
main.o: /usr/include/sys/types.h /usr/include/unistd.h /usr/include/stdlib.h
main.o: Object.h /usr/include/string.h Global.h /usr/include/assert.h
Test.o: Test.h Timer.h /usr/include/sys/time.h /usr/include/standards.h
Test.o: /usr/include/sgidefs.h /usr/include/sys/times.h
15 Test.o: /usr/include/sys/types.h /usr/include/unistd.h /usr/include/stdlib.h
Test.o: Object.h /usr/include/string.h Global.h /usr/include/assert.h
Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h
Timer.o: Timer.h /usr/include/sys/time.h /usr/include/standards.h
20 Timer.o: /usr/include/sgidefs.h /usr/include/sys/times.h
Timer.o: /usr/include/sys/types.h /usr/include/unistd.h /usr/include/stdlib.h
Timer.o: Object.h /usr/include/string.h Global.h /usr/include/assert.h
::::::::::
oop/Timer/Test.C
::::::::::
25 // Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96

#include "Test.h"

30 void Test::Run()          { Timer::Demo(); }

// end of Test.cpp
::::::::::
oop/Timer/Test.h
::::::::::
35 // Test.h      test class for OOP-classes
// (c) Ingmar Bitter '96

#ifndef _Test_h_           // prevent multiple includes
#define _Test_h_
40
#include "Timer.h"

class Test { public: static void Run(); };

#endif      // _Test_h_
45 ::::::::::::::
oop/Timer/Timer.C
::::::::::
// Timer.C
// (c) Ingmar Bitter '96

```

50

55

```

424
// Copyright, Mitsubishi Electric      Information Technology Center
// America, Inc., 1997, All rights reserved.

5   #include "Timer.h"

void Timer::Demo()
{
    // this demo only gives ok results with -O optimizes the loops away)
10  Timer timer;
    cout << endl << "Demo of class " << typeid(timer).name();
    cout << endl << "size : " << sizeof(Timer) << " Bytes";
    cout << endl << "public member functions:";

    long k,n(10000000);
    for (k=0; k<n; ++k);
    cout << endl << " This is a running Timer: " << timer;
    for (k=0; k<n; ++k);
    cout << endl << "... still running: " << timer;
    for (k=0; k<n; ++k);
    cout << endl << " timer.Stop():           " << timer.Stop();
20  for (k=0; k<n; ++k);
    cout << endl << "... still stopped: " << timer;
    cout << endl << " timer.Restart():        " << timer.Restart();
    for (k=0; k<n; ++k);
    cout << endl << " timer.ElapsedTime():     " << timer.ElapsedTime();
25  cout << endl << " timer:                  " << timer;
    cout << endl << "End of demo of class " << typeid(timer).name() << endl;
} // Demo

///////////////////////////////
30 // constructors & destructors

Timer::Timer()
    : timerIsRunning(false)
{
    // start timer
    Restart();
} // constructor

/////////////////////////////
40 // show/set data & data properties
//
// - class Object requirements

ostream & Timer::Ostream(ostream & os) const
{
45    // append Timer info to os
    double t(ElapsedTime());

    double sec (t - int(t) + int(t)%60);
    double min (int(t)/60%60);
    double hour(int(t)/60/60);
50
}

```

```

os << "time:" <<t<< "s = "<<hour<<"h "<<min<<"min "<<sec<<"s";
5 // return complete os
return os;

} // Ostream

10 /////////////////
// show/set data & data properties
//
// - local show/set functions

15 Timer & Timer::Restart()
{
    times(& startTime);
    timerIsRunning = true;
    return *this;
20 } // Restart

double Timer::ElapsedTime() const
{
25     if (!timerIsRunning)
        return runningTime;

        // get current time for time snapshot
        struct tms currentTime, elapsedTime;
        times(& currentTime);
        elapsedTime.tms_utime = currentTime.tms_utime - startTime.tms_utime;
        elapsedTime.tms_stime = currentTime.tms_stime - startTime.tms_stime;
        elapsedTime.tms_cutime = currentTime.tms_cutime - startTime.tms_cutime;
        elapsedTime.tms_cstime = currentTime.tms_cstime - startTime.tms_cstime;
        double seconds = elapsedTime.tms_utime + elapsedTime.tms_stime
            + elapsedTime.tms_cutime+ elapsedTime.tms_cstime;
35     seconds /= (double)CLK_TCK;
        return seconds;
    } // ElapsedTime

40 Timer & Timer::Stop()
{
    runningTime = ElapsedTime();
    timerIsRunning = false;
    return *this;
45 } // Stop

// end of Timer.C
:::::::::::
oop/Timer/Timer.h
:::::::::::
50

```

426

```

// Timer.h
// (c) Ingmar Bitter '96
5
// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

// Timer to track program running time (only real system time, not wall clock
time)
10
#ifndef _Timer_h_ // prevent multiple includes
#define _Timer_h_

15
#include <sys/time.h>
#include <sys/times.h>
#include <unistd.h>

#include "Object.h"

20
class Timer : virtual public Object {
public:
    static void      Demo ();
    // constructors & destructors
    /*inline*/        Timer          ();
25
    // show/set data & data properties
    // - class Object requirements
    virtual /*inline*/ ostream & Ostream (ostream &) const;

    // - local show/set functions
    virtual /*inline*/ Timer & Restart          ();
    virtual /*inline*/ double   ElapsedTime () const;
    virtual /*inline*/ Timer & Stop             ();
protected:
    bocl            timerIsRunning;
35
    double          runningTime;
    struct tms      startTime;
};

#endif      // _Timer_h_
40
:::::::::::
oop/Vector3D/Makefile
:::::::::::
#      Makefile for C++ programs (c) Ingmar Bitter '97
#
#      make           : compile only changed files and their depend
45
files
#      make all       : recompile all
#      make clean    : recompile all and remove unnecessary files
#
EXECUTABLE = go
50

```

55

```

SRC    = main.C Test.C Object.C Vector3D.C
5      RM = /bin/rm -fs

#
#      put -g here | to add debugging info to executable
#          V
10     CDEBUGFLAGS = -O
CCOPTIONS = -64

CC = CC

15     INCLUDES   =
LIBS      = -lm
CFLAGS     = $(CCOPTIONS) $(CDEBUGFLAGS) $(INCLUDES)

OFILES = $(SRC:.C=.o)

20     $(EXECUTABLE): $(OFILES)
                  $(CC) $(OFILES) $(CFLAGS) $(LIBS) -o $(EXECUTABLE)

#
#      target: dependency \n tab rule
#
25     .C.o:
                  $(CC) $(CFLAGS) -c $<

#
#      switch:: ; \n tab rule
#
30     clean:      ;
                  find . -name "*.o" -print -exec mv {} ~/dumpster \; ;
                  find . -name "*" -print -exec mv {} ~/dumpster \; ;
                  find . -name "go" -print -exec mv {} ~/dumpster \; ;
                  find . -name "core" -print -exec mv {} ~/dumpster \; ;

35     all:      ;
                  pmake -u

depend:
                  makedepend -- $(CFLAGS) -- $(SRC)

40     # DO NOT DELETE

main.o: Test.h Vector3D.h /usr/include/math.h /usr/include/sgidefs.h
main.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
main.o: /usr/include/assert.h
45     Test.o: Test.h Vector3D.h /usr/include/math.h /usr/include/sgidefs.h
Test.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Test.o: /usr/include/assert.h
Object.o: Object.h /usr/include/string.h /usr/include/standards.h Global.h
Object.o: /usr/include/assert.h
50

```

```

428
Vector3D.o: Vector3D.h           /usr/include/math.h
/usr/include/sgidefs.h
Vector3D.o: /usr/include/standards.h Object.h /usr/include/string.h Global.h
Vector3D.o: /usr/include/assert.h
:::::::::::
oop/Vector3D/Test.C
:::::::::::
// Test.cpp      testclass for OOP-classes
// (c) Ingmar Bitter '96
10

#include "Test.h"

void Test::Run() { Vector3D<int>::Demo(); }

15 // end of Test.cpp
:::::::::::
oop/Vector3D/Test.h
:::::::::::
// Test.h      test class for OOP-classes
// (c) Ingmar Bitter '96
20

#ifndef _Test_h_          // prevent multiple includes
#define _Test_h_

#include "Vector3D.h"

25 class Test { public: static void Run(); };

#endif      // _Test_h_
:::::::::::
oop/Vector3D/Vector3D.C
:::::::::::
30 // Vector3D.C
// (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

35 #include "Vector3D.h"

template <class T> void Vector3D<T>::Demo()
{
    Vector3D<T> a, b(3.5,2.5,1.5);
    cout << endl << "Demo of class " << typeid(a).name();
40    cout << endl << "size : " << sizeof(Vector3D<T>)/8 << " Bytes";
    cout << endl << "public member functions:";
    cout << endl << "  <<typeid(a).name()<< a, b(3.5,2.5,1.5); == " << a << ", <<
&b;
    cout << endl << "  a<b; \t== " << (a<b) << "\t a<=b; \t== " <<
45    (a<=b)<<"\ta==b; \t== " << (a==b);
        cout << endl << "  a>b; \t== " << (a>b) << "\t a>=b; \t== " <<
(a>=b)<<"\ta!=b; \t== " << (a!=b);

```

50

55

```

429
cout << endl << " a=b;\t== " << (a=b);
cout << endl << " a.X();\t== " << (a.X()) << "\t a.U();\t== " << (a.U());
cout << endl << " a.Y();\t== " << (a.Y()) << "\t a.V();\t== " << (a.V());
cout << endl << " a.Z();\t== " << (a.Z()) << "\t a.W();\t== " << (a.W());

cout << endl << " +a;\t== " << (+a) << "\t a+b;\t== " << (a+b);
cout << endl << " -a;\t== " << (-a) << "\t a-b;\t== " << (a-b);
cout << endl << " a+=b;\t== " << (a+=b);
cout << endl << " a-=b;\t== " << (a-=b);
T s; s = (T) 2.5;
cout << endl << " <<typeid(s).name()<< s(2.5);\t== " << s;
cout << endl << " a*s;\t== " << (a*s) << "\t a*=s;\t== " << (a*=s);
cout << endl << " a/s;\t== " << (a/s) << "\t a/=s;\t== " << (a/=s);
cout << endl << " a.DotProduct(b);\t== " << a.DotProduct(b);
cout << endl << " a.Dot(b);\t\t== " << a.Dot(b);
cout << endl << " a*b;\t\t\t== " << (a*b);
cout << endl << " a.CrossProduct(b);\t\t== " << a.CrossProduct(b);
cout << endl << " a.Cross(b);\t\t\t== " << a.Cross(b);
cout << endl << " a.X(b);\t\t\t== " << a.X(b);
cout << endl << " a/b;\t\t\t\t== " << (a/b);

20 cout << endl << "End of demo of class " << typeid(a).name() << endl;
} // Demo

25 //////////////////////////////////////////////////////////////////
// constructors & destructors

30 template <class T> Vector3D<T>::Vector3D(const T & x, const T & y, const T & z)
{
    element[0] = x; element[1] = y; element[2] = z;
} // constructor

35 template <class T> Vector3D<T>::Vector3D(const Vector3D<T> & v)
{
    element[0] = v.element[0]; element[1] = v.element[1]; element[2] =
v.element[2];
} // copy-constructor

40 template <class T> Vector3D<T>::Vector3D(Vector3D<int> & v)
{
    element[0] = v.X(); element[1] = v.Y(); element[2] = v.Z();
} // copy-constructor

45 template <class T> Vector3D<T>::Vector3D(Vector3D<ModInt> & v)
{
    element[0] = v.X(); element[1] = v.Y(); element[2] = v.Z();
} // copy-constructor

```

```

////////// casts to other data types
5 // casts to other data types

template <class T> Vector3D<T>::operator double() const { return Norm(); }

////////// show/set data & data properties
10 // -
// - class Object requirements

template <class T> ostream & Vector3D<T>::Ostream(ostream & os) const
{
    return os<<"("<<X()<<","<<Y()<<","<<Z()<<")";
} // Ostream

////////// show/set data & data properties
20 // -
// - class CmpObj requirements

template <class T> double Vector3D<T>::Norm() const
{
    return sqrt(X()*X() + Y()*Y() + Z()*Z());
} // Norm

////////// local show/set functions
30 // -
// - local show/set functions

template <class T> T Vector3D<T>::X() const { return element[0]; }
template <class T> T Vector3D<T>::Y() const { return element[1]; }
template <class T> T Vector3D<T>::Z() const { return element[2]; }

template <class T> T Vector3D<T>::U() const { return element[0]; }
template <class T> T Vector3D<T>::V() const { return element[1]; }
40 template <class T> T Vector3D<T>::W() const { return element[2]; }

template <class T> T Vector3D<T>::R() const { return element[0]; }
template <class T> T Vector3D<T>::G() const { return element[1]; }
template <class T> T Vector3D<T>::B() const { return element[2]; }

45 template <class T> Vector3D<T> & Vector3D<T>::SetX(const T & setX) {
    element[0]=setX; return *this;
}
template <class T> Vector3D<T> & Vector3D<T>::SetY(const T & setY) {
    element[1]=setY; return *this;
}
50 template <class T> Vector3D<T> & Vector3D<T>::SetZ(const T & setZ) {
    element[2]=setZ; return *this;
}
```

```

        431
template <class T> Vector3D<T> &           Vector3D<T>::SetU(const T & setU) {
5   element[0]=setU; return *this; }
template <class T> Vector3D<T> & Vector3D<T>::SetV(const T & setV) {
element[1]=setV; return *this; }
template <class T> Vector3D<T> & Vector3D<T>::SetW(const T & setW) {
element[2]=setW; return *this; }

10  template <class T> Vector3D<T> & Vector3D<T>::SetR(const T & setR) {
element[0]=setR; return *this; }
template <class T> Vector3D<T> & Vector3D<T>::SetG(const T & setG) {
element[1]=setG; return *this; }
template <class T> Vector3D<T> & Vector3D<T>::SetB(const T & setB) {
element[2]=setB; return *this; }

15  template <class T> Vector3D<T> & Vector3D<T>::operator () (const T & x, const T
& y, const T & z)
{
    element[0] = x; element[1] = y; element[2] = z;
    return *this;
} // operator ()

20

template <class T> T Vector3D<T>::operator [] (const int index) const
{
    return element[index];
} // operator []

25

template <class T> bool Vector3D<T>::operator ==(const Vector3D<T> & a) const {
return ( X()==a.X() && Y()==a.Y() && Z()==a.Z() ); }
template <class T> bool Vector3D<T>::operator !=(const Vector3D<T> & a) const {
return ( X()!=a.X() || Y()!=a.Y() || Z()!=a.Z() ); }

30

template <class T> Vector3D<T> Vector3D<T>::operator + () const { return
Vector3D<T>(*this); }
template <class T> Vector3D<T> Vector3D<T>::operator - () const { return
Vector3D<T>(-element[0], -element[1], -element[2]); }

35

template <class T> Vector3D<T> Vector3D<T>::operator + (const Vector3D<T> & a)
const { return Vector3D<T>(X()+a.X(), Y()+a.Y(), Z()+a.Z()); }
template <class T> Vector3D<T> Vector3D<T>::operator - (const Vector3D<T> & a)
const { return Vector3D<T>(X()-a.X(), Y()-a.Y(), Z()-a.Z()); }
template <class T> Vector3D<T> & Vector3D<T>::operator +=(const Vector3D<T> & a)
40
{ element[0]+=a.X(); element[1]+=a.Y(); element[2]+=a.Z(); return *this; }
template <class T> Vector3D<T> & Vector3D<T>::operator -=(const Vector3D<T> & a)
{ element[0]-=a.X(); element[1]-=a.Y(); element[2]-=a.Z(); return *this; }

45

template <class T> Vector3D<T> Vector3D<T>::operator * (const T & a) const {
return Vector3D<T>(X()*a, Y()*a, Z()*a); }
template <class T> Vector3D<T> Vector3D<T>::operator / (const T & a) const {
return Vector3D<T>(X()/a, Y()/a, Z()/a); }
template <class T> Vector3D<T> & Vector3D<T>::operator *=(const T & a)
50
{ element[0]*=a; element[1]*=a; element[2]*=a; return *this; }

```

50

55

```

432
template <class T> Vector3D<T> &           Vector3D<T>::operator /=(const T & a)
{ element[0]/=a; element[1]/=a; element[2]/=a; return *this; }

5   template <class T> T           Vector3D<T>::DotProduct (const Vector3D<T> &
a) const { return *this * a; }
template <class T> T           Vector3D<T>::Dot      (const Vector3D<T> &
a) const { return *this * a; }
template <class T> T           Vector3D<T>::operator * (const Vector3D<T> &
10  a) const { return (X()*a.X() + Y()*a.Y() + Z()*a.Z()); }

template <class T> Vector3D<T>  Vector3D<T>::CrossProduct(const Vector3D<T> &
a) const { return *this / a; }
template <class T> Vector3D<T>  Vector3D<T>::Cross      (const Vector3D<T> &
15  a) const { return *this / a; }
template <class T> Vector3D<T>  Vector3D<T>::X          (const Vector3D<T> &
a) const { return *this / a; }
template <class T> Vector3D<T>  Vector3D<T>::operator / (const Vector3D<T> &
a) const
{ return Vector3D(Y()*a.Z() - Z()*a.Y(), Z()*a.X() - X()*a.Z(), X()*a.Y() -
20  Y()*a.X()); }

// end of Vector3D.C
:::::::::::
oop/Vector3D/Vector3D.h
:::::::::::
25 // Vector3D.h
// (c) Ingmar Bitter '96

// Copyright, Mitsubishi Electric Information Technology Center
// America, Inc., 1997, All rights reserved.

30 // dynamic array ADT

#ifndef _Vector3D_h_    // prevent multiple includes
#define _Vector3D_h_

35 #include <math.h>
#include "Object.h"
#include "ModInt.h"

template <class T> class Vector3D : public virtual Object {
40 public:
    static void      Demo      ();
    // constructors & destructors
    Vector3D (const T & x=0, const T & y=0, const T & z=0);
    Vector3D (const Vector3D<T> &);

45    Vector3D (    Vector3D<int> &);
    Vector3D (    Vector3D<ModInt> &);

    // casts to other data types
    virtual /*inline*/ operator double() const;
}

50

```

55

```

// show/set data & data properties
// - class Object requirements
5    virtual      ostream &   Ostream (ostream & ) const;

// - class CmpObj requirements
virtual /*inline*/ double Norm () const;

10   // - local show/set functions
virtual /*inline*/ T X () const;
virtual /*inline*/ T Y () const;
virtual /*inline*/ T Z () const;

15   virtual /*inline*/ T U () const;
virtual /*inline*/ T V () const;
virtual /*inline*/ T W () const;

virtual /*inline*/ T R () const;
virtual /*inline*/ T G () const;
20   virtual /*inline*/ T B () const;

virtual /*inline*/ Vector3D<T> & SetX (const T &);
virtual /*inline*/ Vector3D<T> & SetY (const T &);
virtual /*inline*/ Vector3D<T> & SetZ (const T &);

25   virtual /*inline*/ Vector3D<T> & SetU (const T &);
virtual /*inline*/ Vector3D<T> & SetV (const T &);
virtual /*inline*/ Vector3D<T> & SetW (const T &);

30   virtual /*inline*/ Vector3D<T> & SetR (const T &);
virtual /*inline*/ Vector3D<T> & SetG (const T &);
virtual /*inline*/ Vector3D<T> & SetB (const T &);

35   virtual /*inline*/ Vector3D<T> & operator () (const T & x, const T & y,
:const T & z);
    virtual /*inline*/ T operator [] (const int index) const;

// local computation functions
virtual /*inline*/ bool operator == (const Vector3D<T> &) const;
virtual /*inline*/ bool operator != (const Vector3D<T> &) const;

40   virtual /*inline*/ Vector3D<T> operator + ( ) const;
virtual /*inline*/ Vector3D<T> operator - ( ) const;

virtual /*inline*/ Vector3D<T> operator + (const Vector3D<T> &) const;
virtual /*inline*/ Vector3D<T> operator - (const Vector3D<T> &) const;
virtual /*inline*/ Vector3D<T> & operator += (const Vector3D<T> &);
45   virtual /*inline*/ Vector3D<T> & operator -= (const Vector3D<T> &);

virtual /*inline*/ Vector3D<T> operator * (const T &) const;
virtual /*inline*/ Vector3D<T> operator / (const T &) const;
virtual /*inline*/ Vector3D<T> & operator *= (const T &);

50

```

```

434
virtual /*inline*/ Vector3D<T> & operator /= (const T &);

5   virtual /*inline*/ T           DotProduct (const Vector3D<T> &) const;
virtual /*inline*/ T           Dot      (const Vector3D<T> &) const;
virtual /*inline*/ T           operator * (const Vector3D<T> &) const;

10  virtual /*inline*/ Vector3D<T> CrossProduct (const Vector3D<T> &) const;
virtual /*inline*/ Vector3D<T> Cross    (const Vector3D<T> &) const;
virtual /*inline*/ Vector3D<T> X        (const Vector3D<T> &) const;
virtual /*inline*/ Vector3D<T> operator / (const Vector3D<T> &) const;

protected:
    T element[3];
15
}; // class Vector3D

#endif // _Vector3D_h_

```

20

Claims

25

1. Apparatus to enable real-time volume rendering, comprising:

a volume graphics rendering engine having a number of memory modules for storing an incoming volume data set and a like number of processing modules connected to respective memory modules and interconnected in a ring-like fashion; and,

means for grouping voxels in said volume data set in such a way that said voxels from said groups can be fetched from consecutive memory addresses in said memory modules regardless of view direction, thereby to permit the use of memory modules of types that support high speed accesses from consecutive memory addresses.

2. The apparatus of Claim 1 wherein said volume graphics rendering engine is applied to a personal computer.
3. The apparatus of Claim 1 wherein said volume graphics rendering engine is coupled to a desktop computer.
- 40 4. The apparatus of Claim 1 wherein said memory modules are Dynamic Random Access Memory modules equipped with the capability for accessing data from consecutive memory addresses in burst mode.
5. The apparatus of Claim 1 wherein said grouping means including means for grouping said voxels into blocks of contiguous voxels in three dimensions and means for skewing said blocks across said memory modules in such a way that adjacent blocks in any of the three dimensions of the volume data set are stored in adjacent memory modules to make possible for adjacent processing modules to concurrently and simultaneously fetch voxels of adjacent blocks in burst mode from respective adjacent memory modules.
- 50 6. The apparatus of Claim 5 wherein all of the voxels in a block are stored in the same memory module.
7. The apparatus of Claim 6 wherein within each block voxels are stored at consecutive memory addresses.
8. The apparatus of Claim 7 wherein said processing modules include means for rendering voxels within a block;

means for forwarding data to adjacent processing modules; and,

means for accumulating results of rendering individual blocks to produce resulting images.

55

9. The apparatus of Claim 8 and further including means for bi-directional communication between adjacent processing modules; and,

means for determining direction of communication for said forwarding data.

5 10. The apparatus of Claim 5 wherein said blocks are grouped into slices such that all blocks of a slice are equidistant from a face of the volume data set.

10 11. The apparatus of Claim 5 and further including means for rendering of said volume data set proceeds in groups of P adjacent blocks at a time, where P is the number of said processing modules in said volume graphics engine:

means for controlling the sequence of said rendering in groups of P blocks at a time so that all blocks of any slice of said volume data said are rendered before any blocks of any other slice more distant from the face of said volume data set than said slice; and,

15 means for accumulating results of rendering blocks a slice at a time.

12. The apparatus of Claim 1 and further including means for partitioning the volume data set into sections;

20 means for rendering said volume data set a section at a time; and,

means for storing results of rendered sections and for reapplying said results to the rendering of subsequent sections.

25 13. The apparatus of Claim 1 and further including means for reducing the amount of internal storage within each of said processing modules including means for sectioning said volume data set thus to permit rendering a section at a time, with each section requiring less storage than that associated with rendering the volume as a whole.

30 14. The apparatus of Claim 1 wherein said memory modules are Dynamic Random Access Memory semi-conductor arrays embedded within respective processing modules on one semi-conductor chip.

15. The apparatus of Claim 5 and further including means for transmitting data only from voxels on the face of each block to the adjacent processing pipeline, thereby to reduce by a factor of 1/B the amount of data transmitted between adjacent pipelines, where B is the number of voxels on an edge of a block.

35 16. The apparatus of Claim 7 and further including means for subdividing said blocks into mini-blocks such that all of the voxels of a mini-block are stored at consecutive memory addresses, thereby to reduce the amount of storage required in said processing modules.

40 17. The apparatus of Claim 13 and further including means for subdividing said blocks into mini-blocks such that all of the voxels of a mini-block are stored at consecutive memory addresses, thereby to reduce the amount of storage required in said processing modules, and means for reducing the amount of time needed to render a section including means for reducing the number of voxels needing to be read from a block at the boundary of said section from all of the voxels of the block at said boundary to only those voxels in mini-blocks at said boundary.

45

50

55

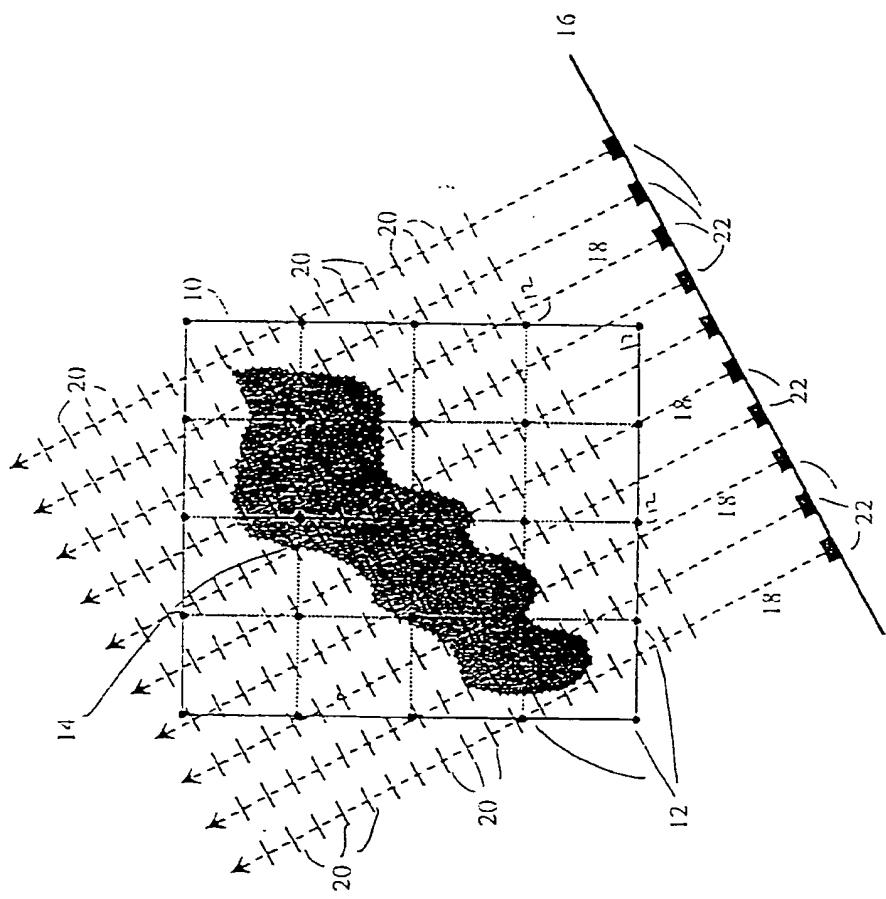


FIG. 1

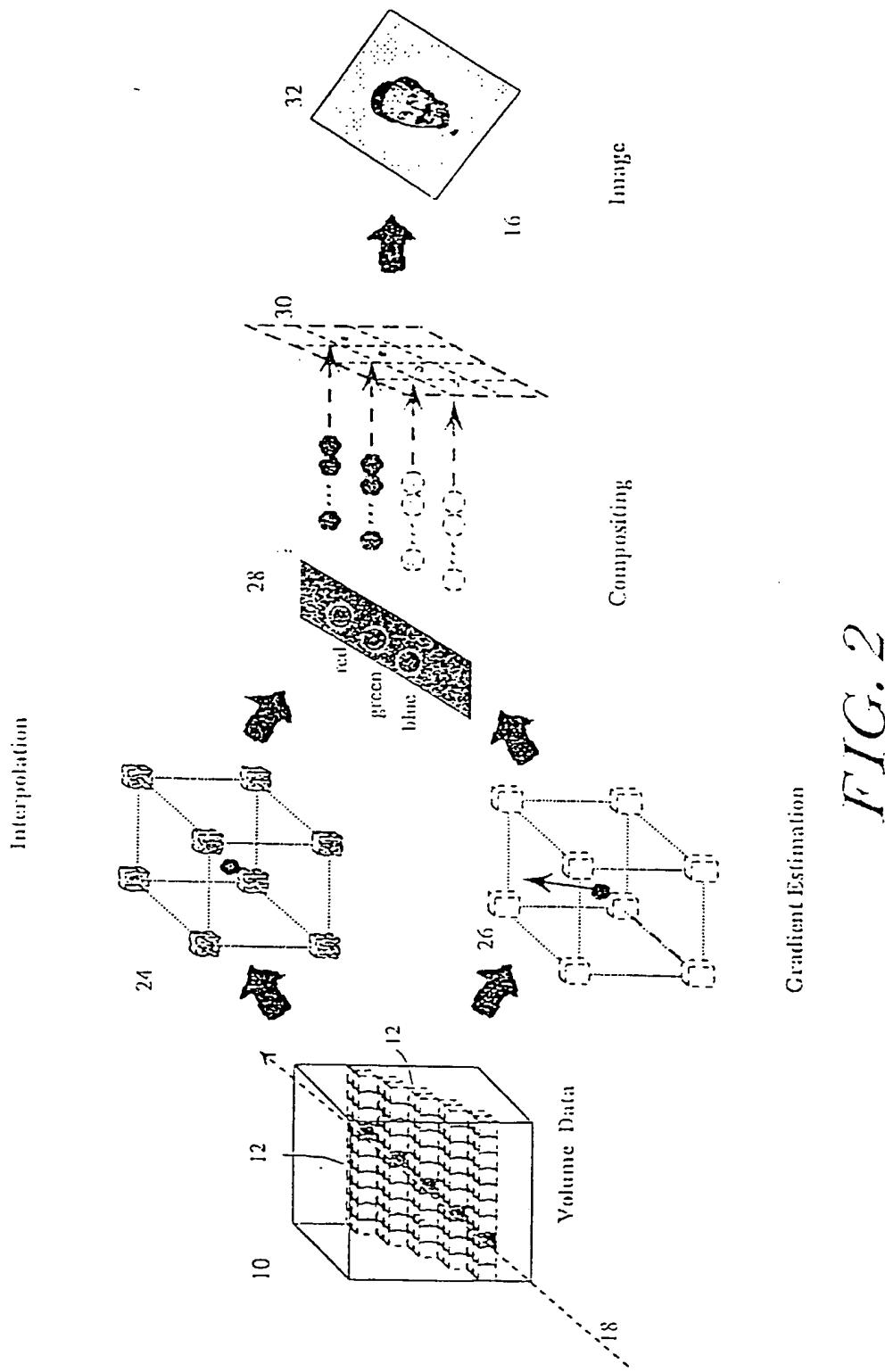


FIG. 2

Gradient Estimation

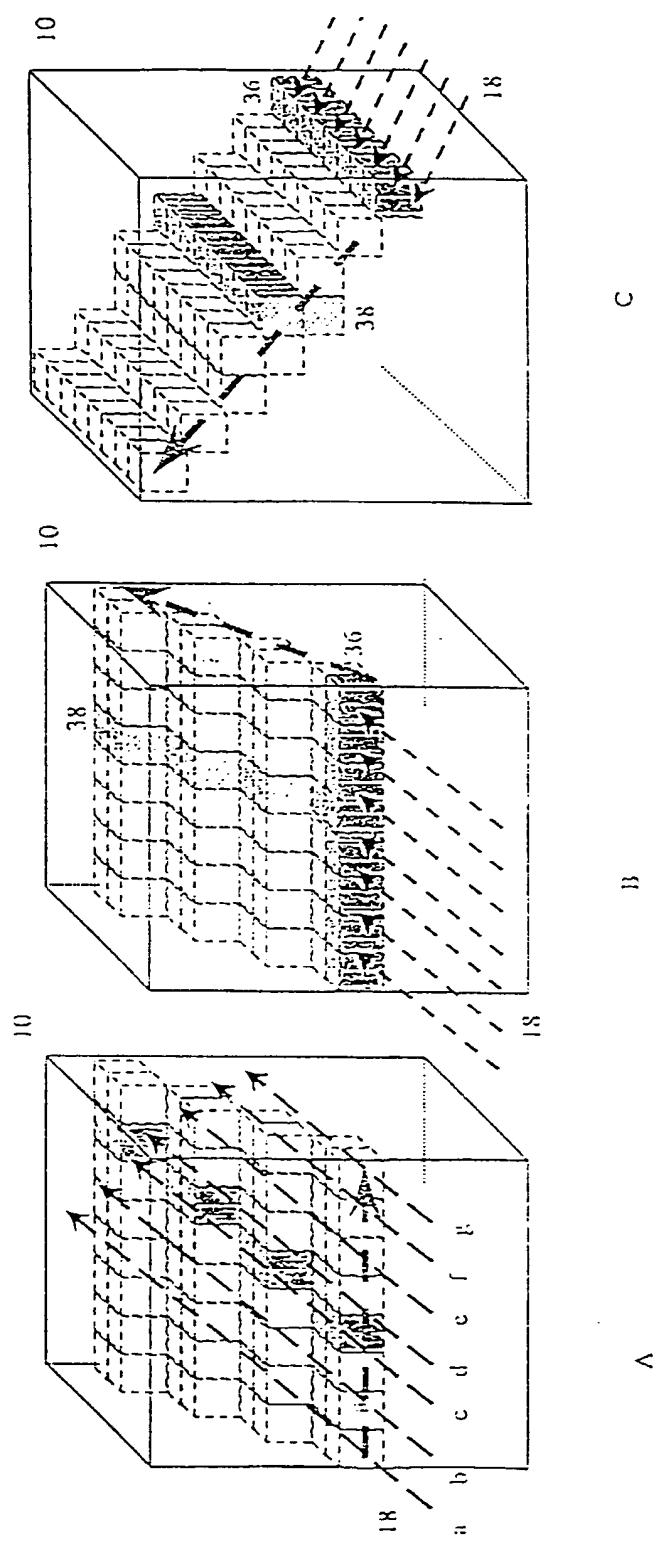


FIG. 3

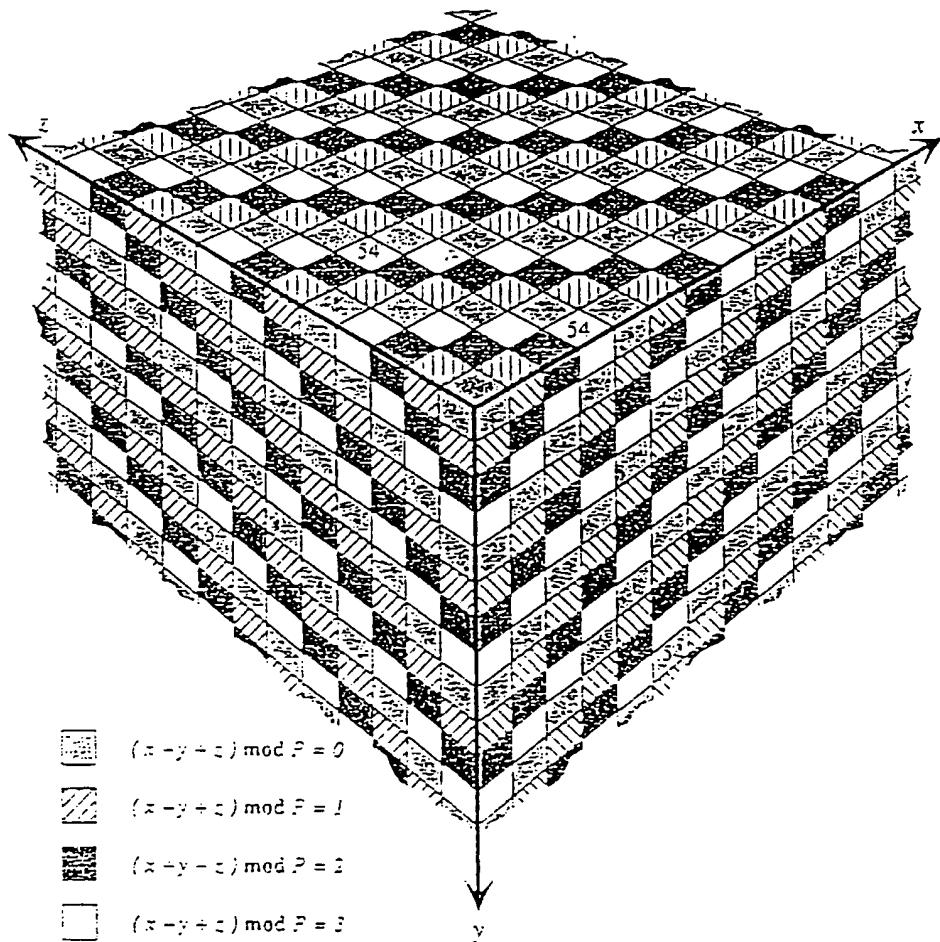


FIG. 4

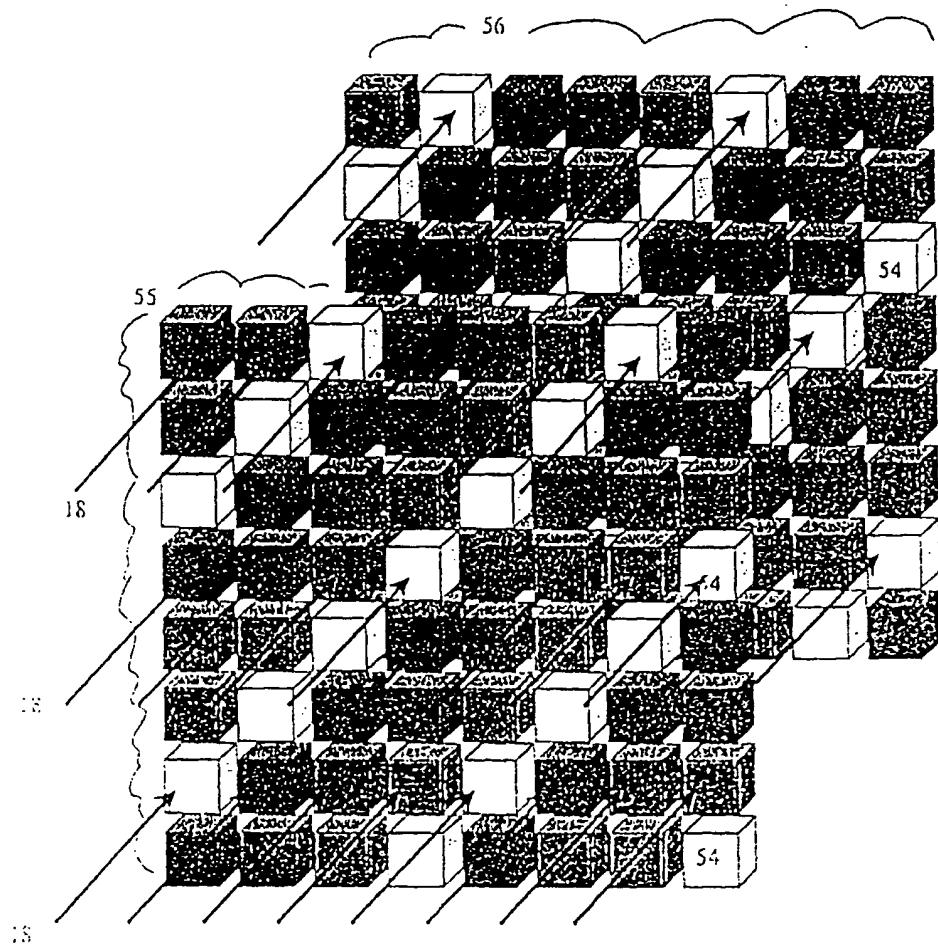


FIG. 5

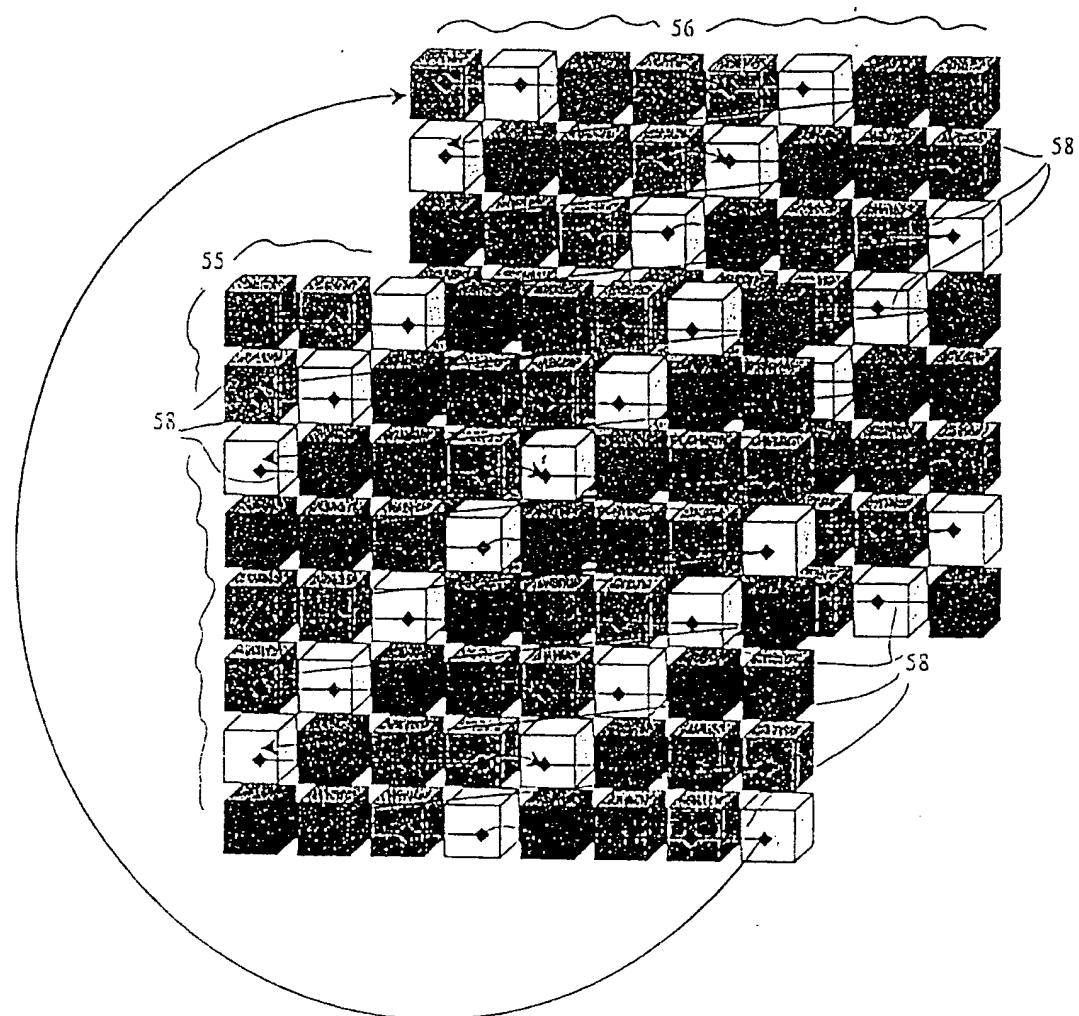


FIG. 6

52

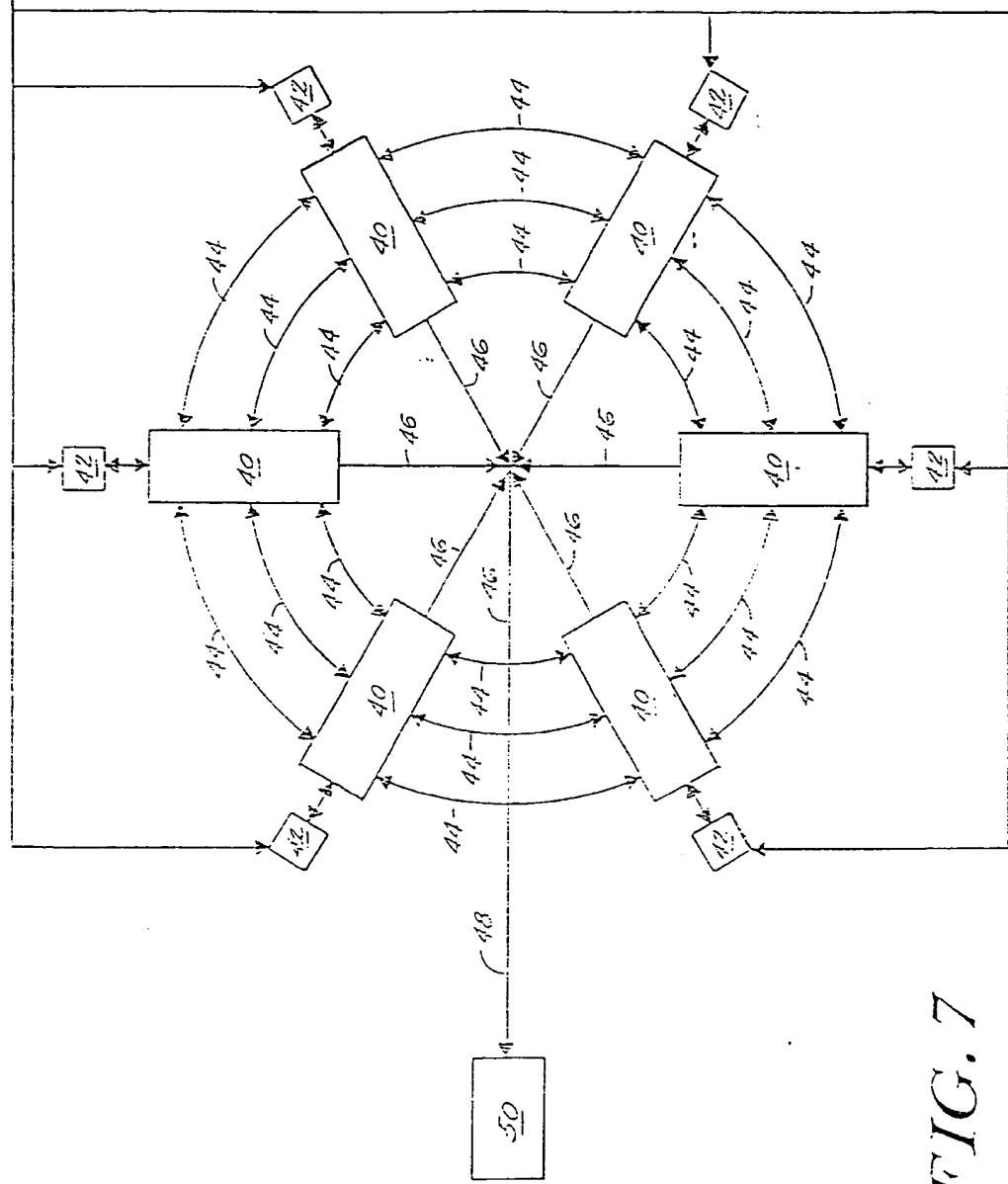
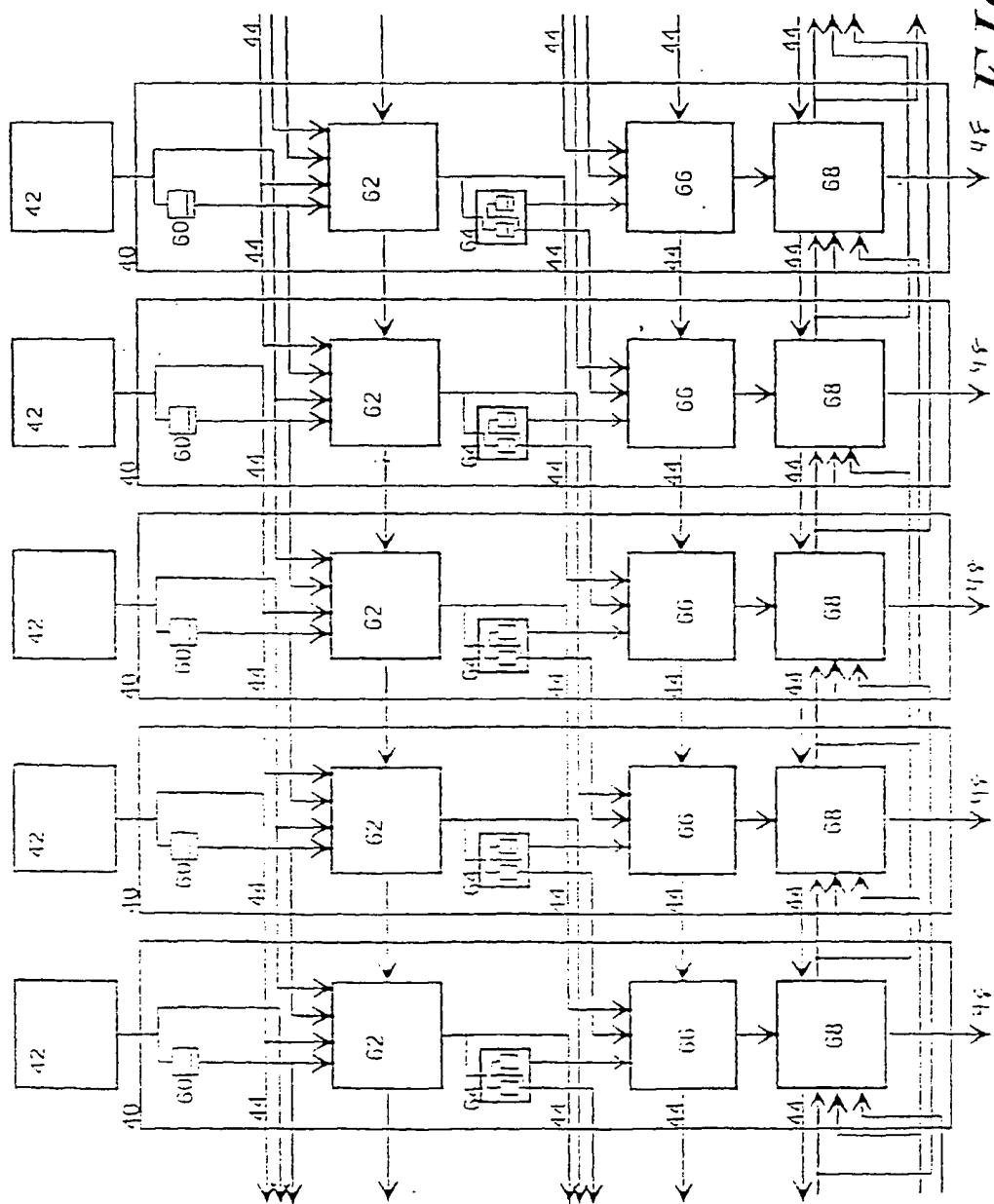


FIG. 7

FIG. 8



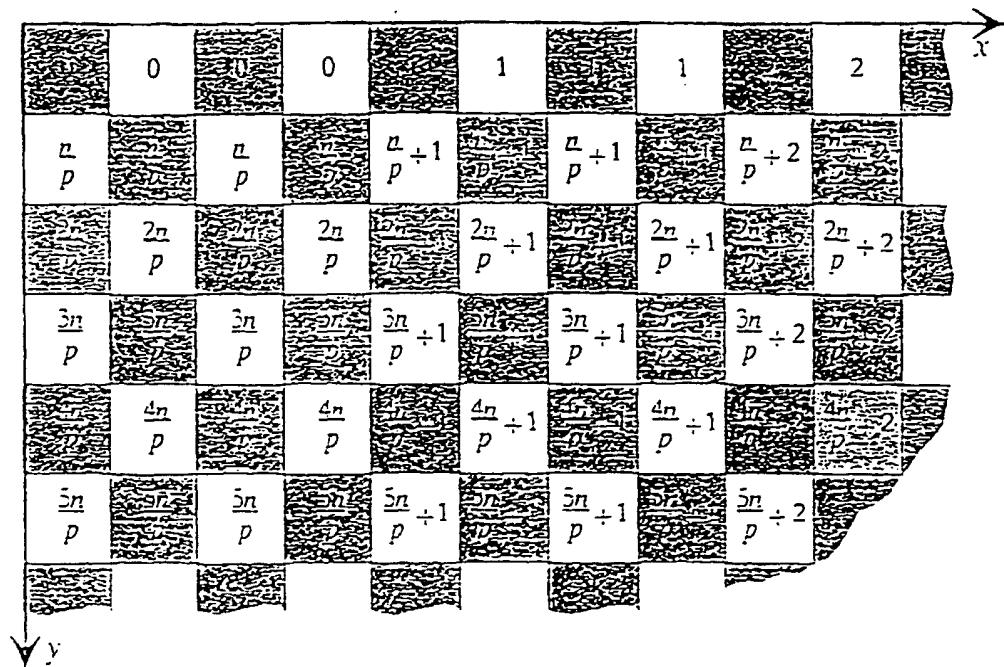


FIG. 9

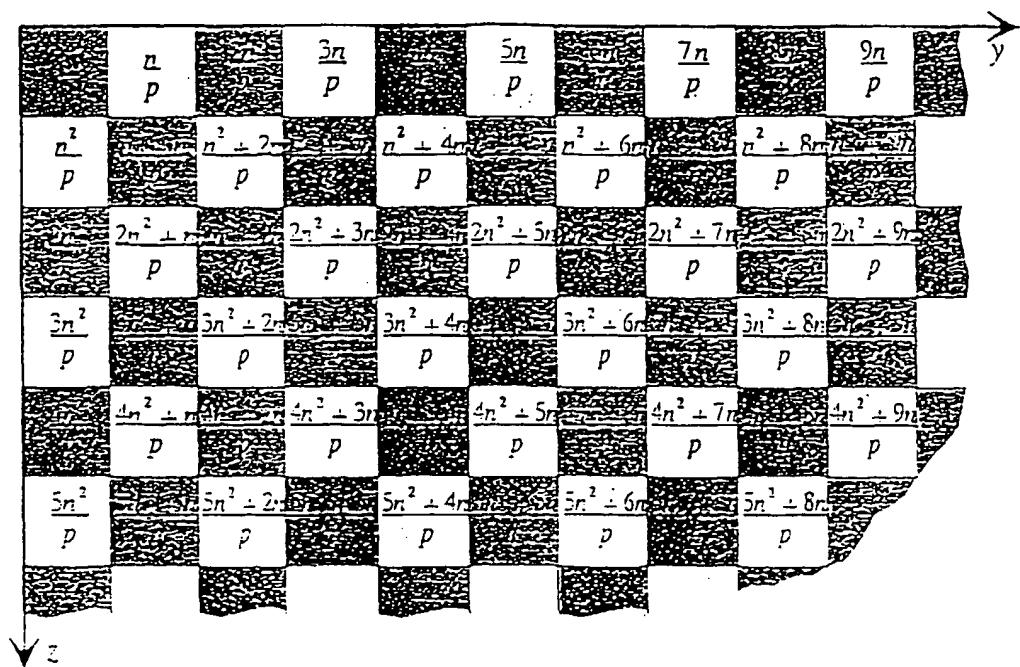


FIG. 10

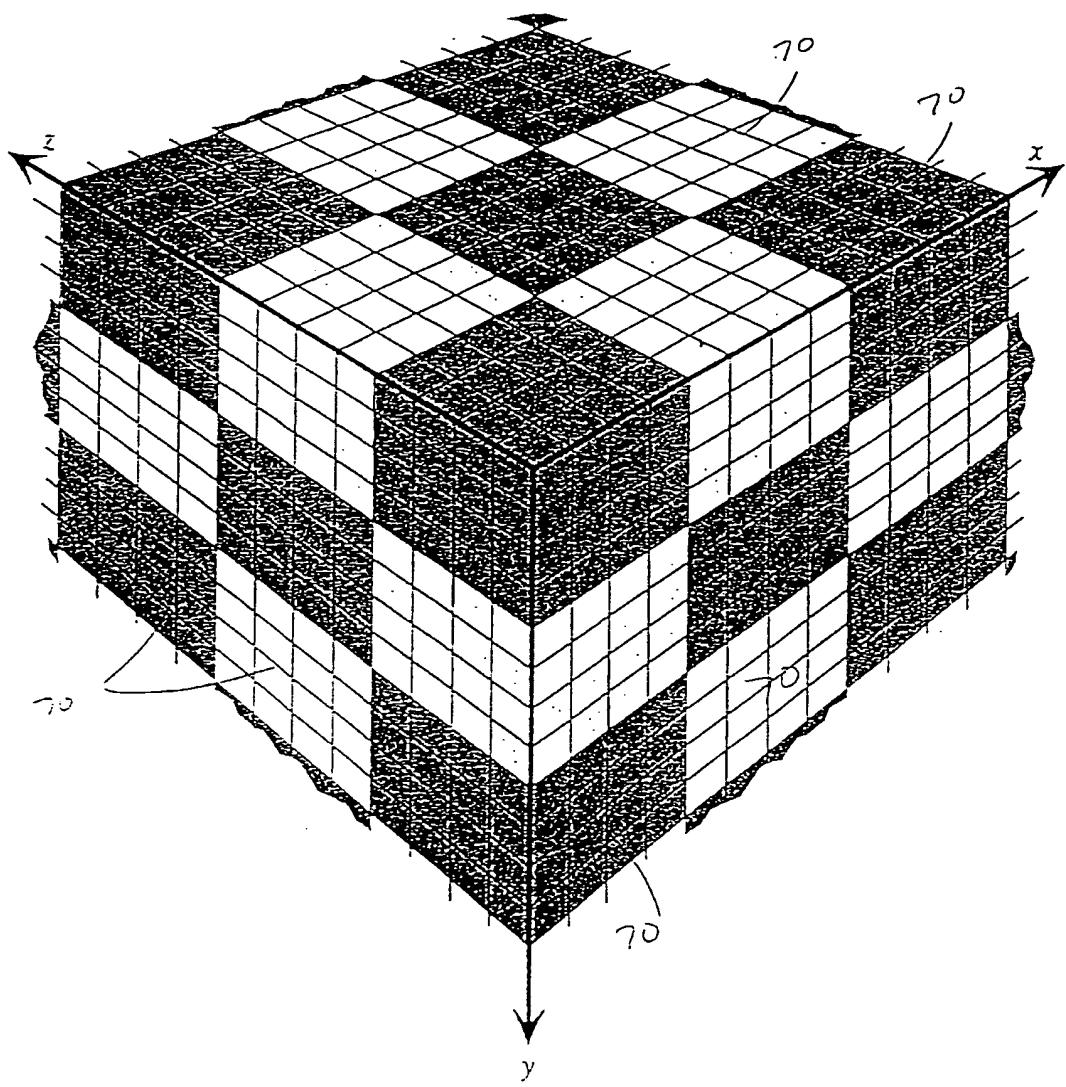


FIG. II

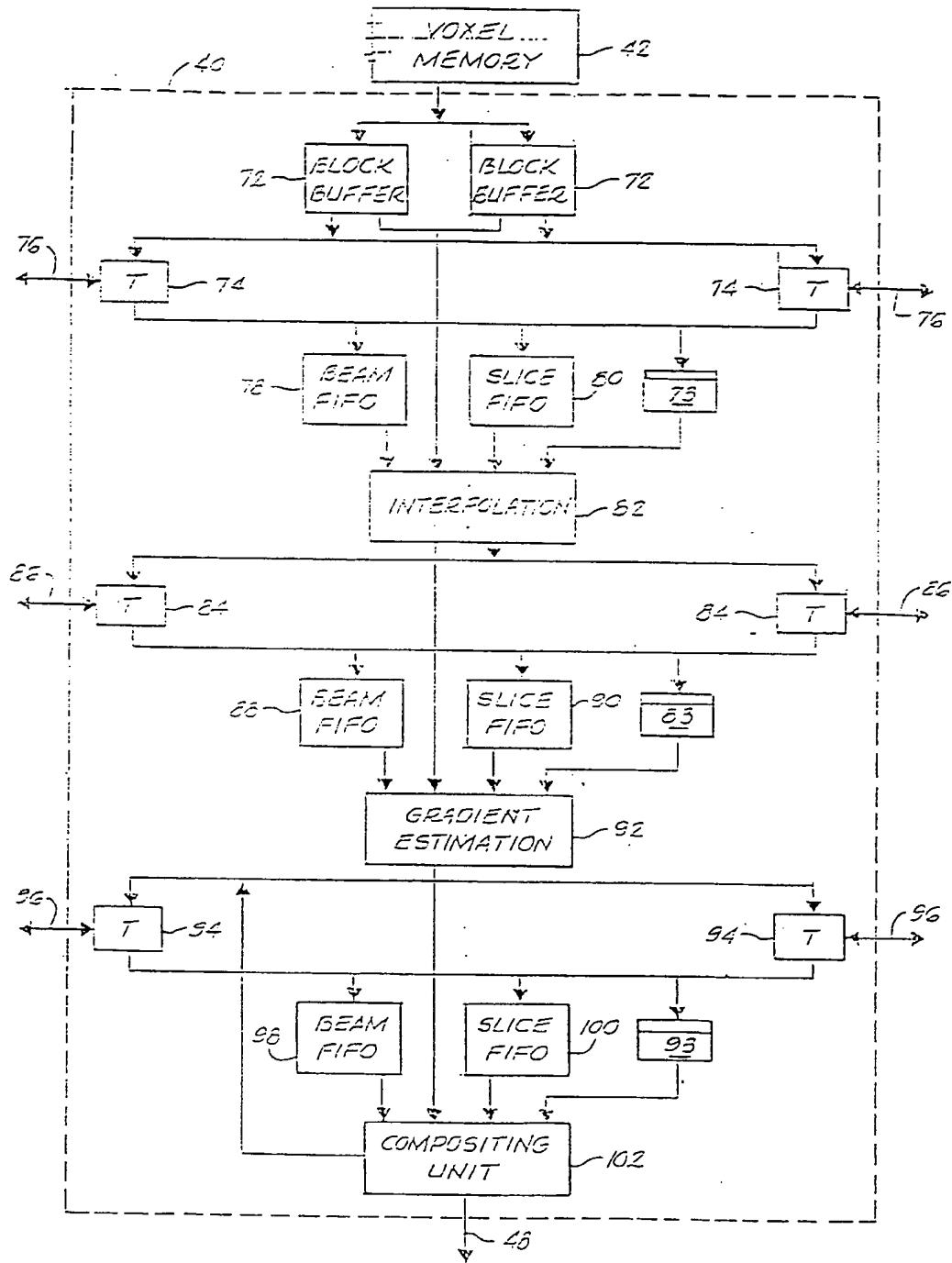


FIG. 12

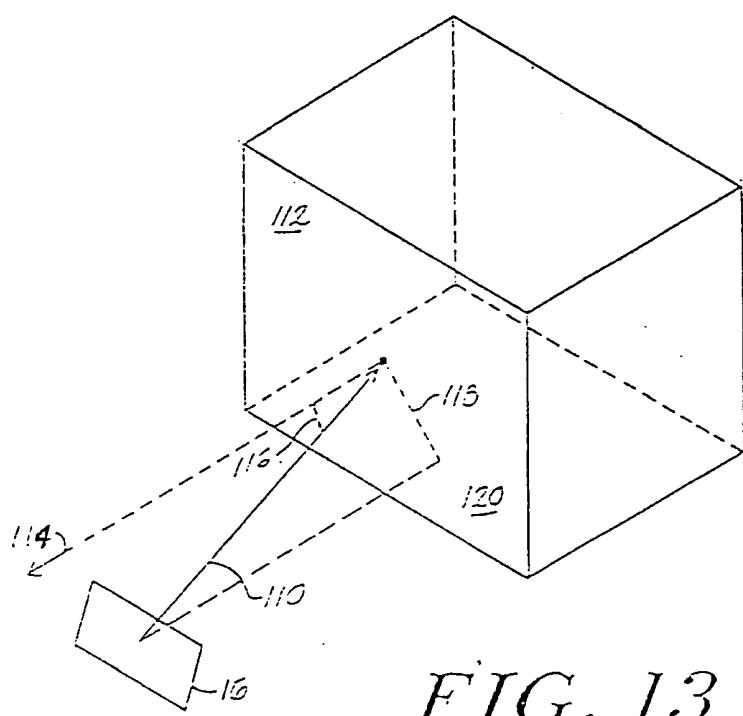
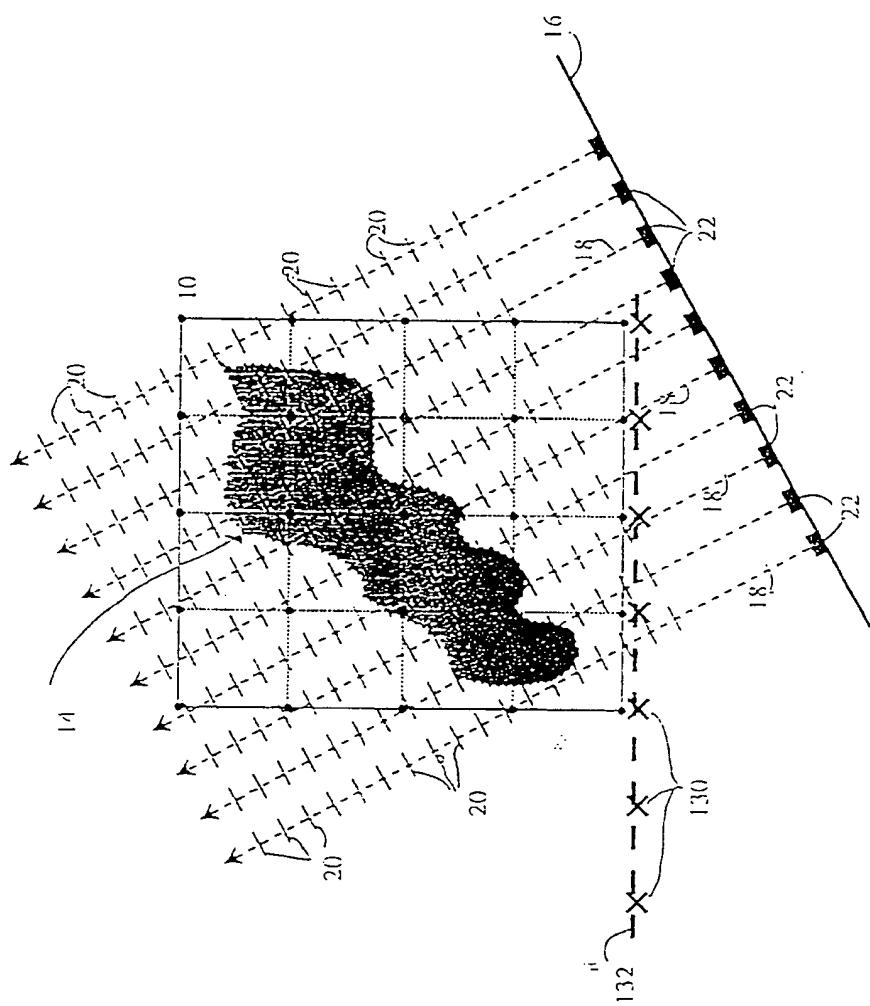
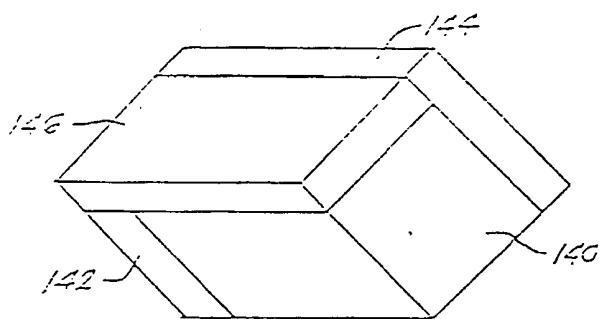
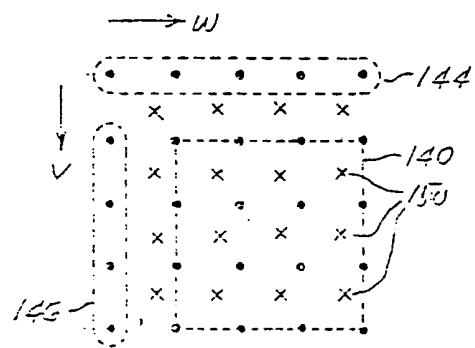
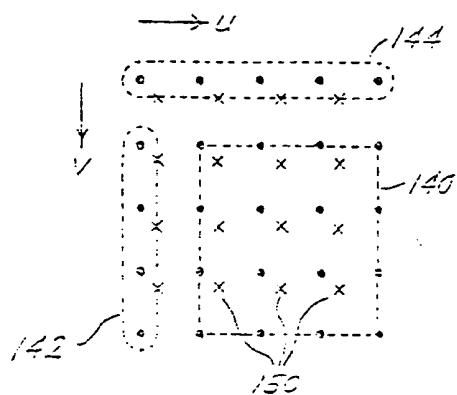


FIG. 13

FIG. 14





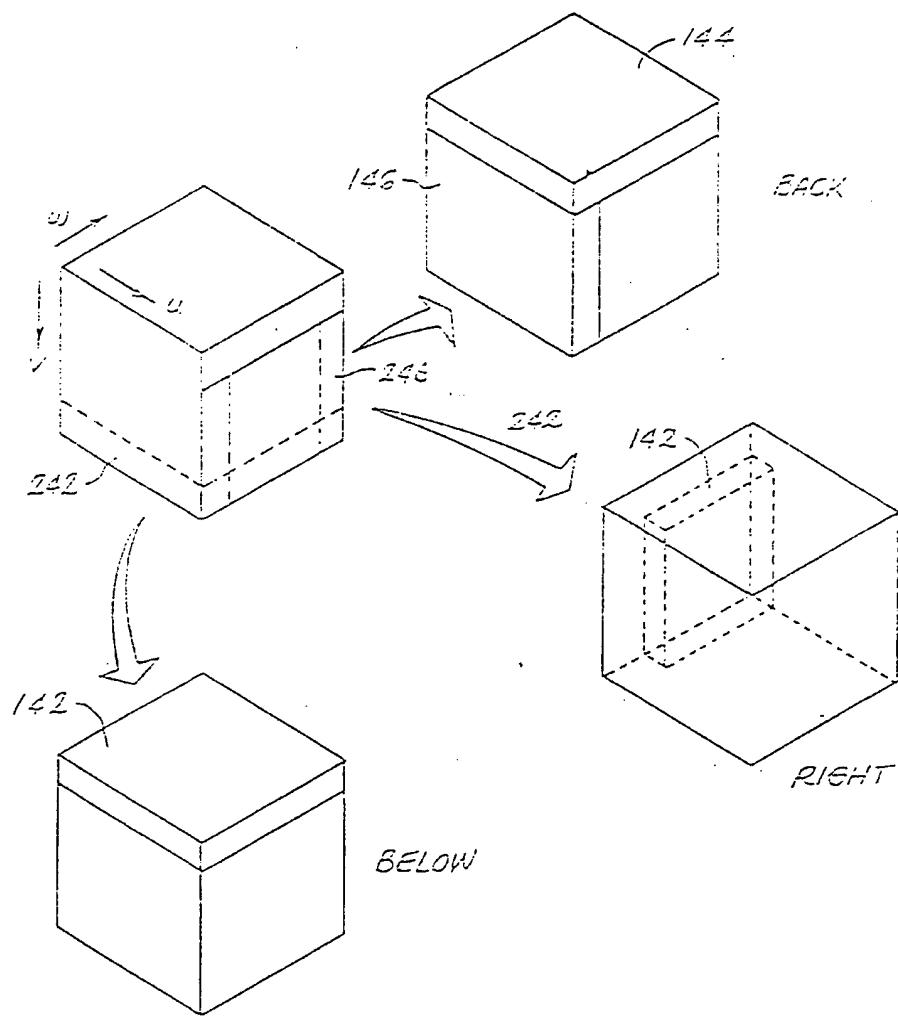


FIG. 16

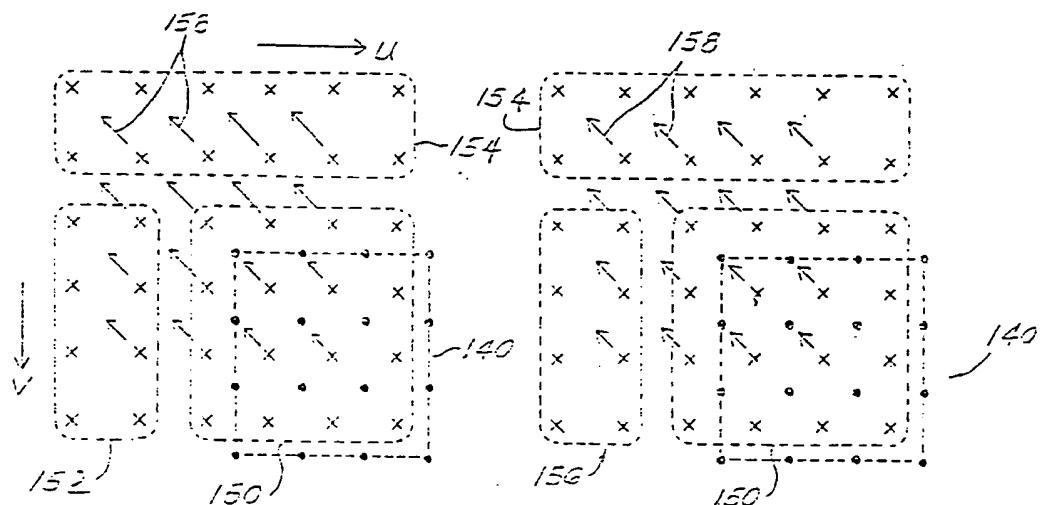


FIG. 17

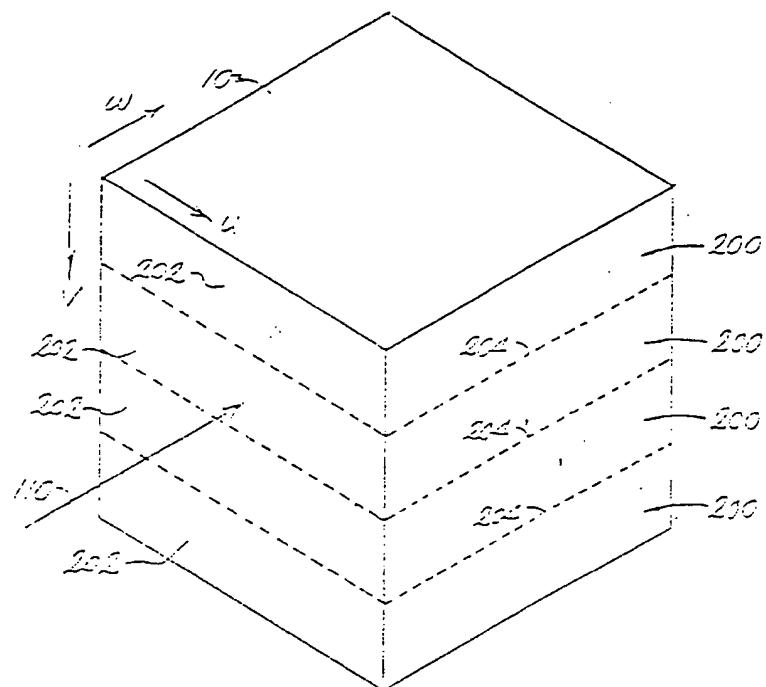


FIG. 18A

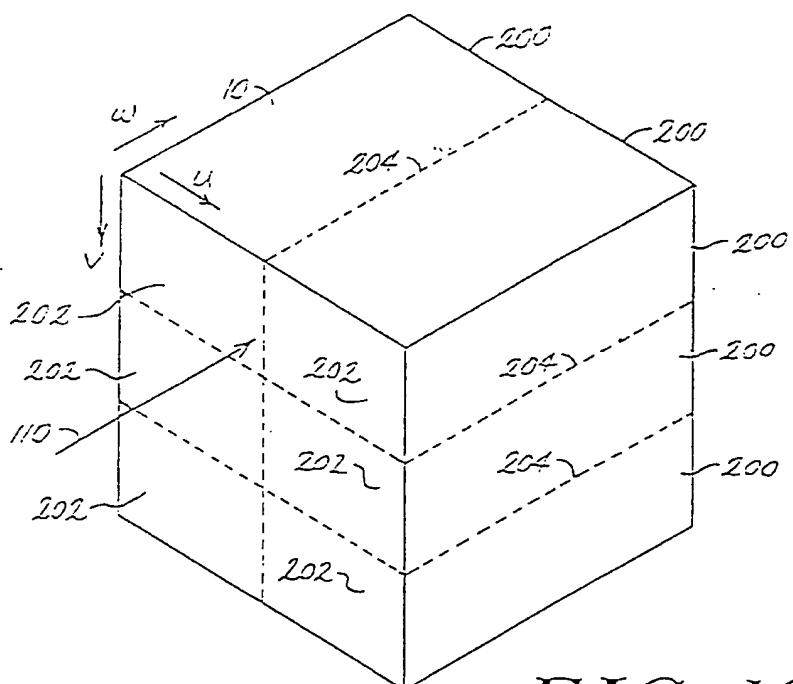


FIG. 18B

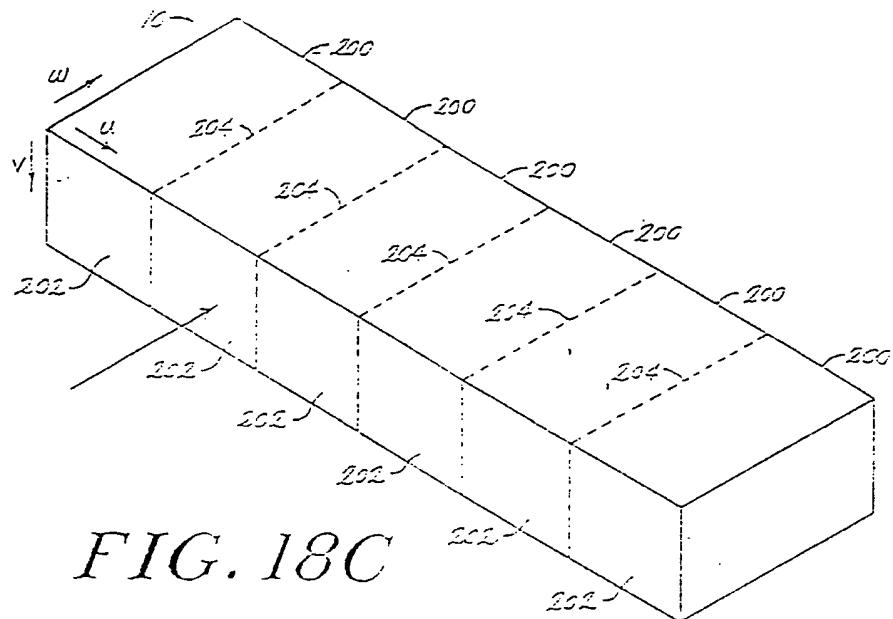


FIG. 18C

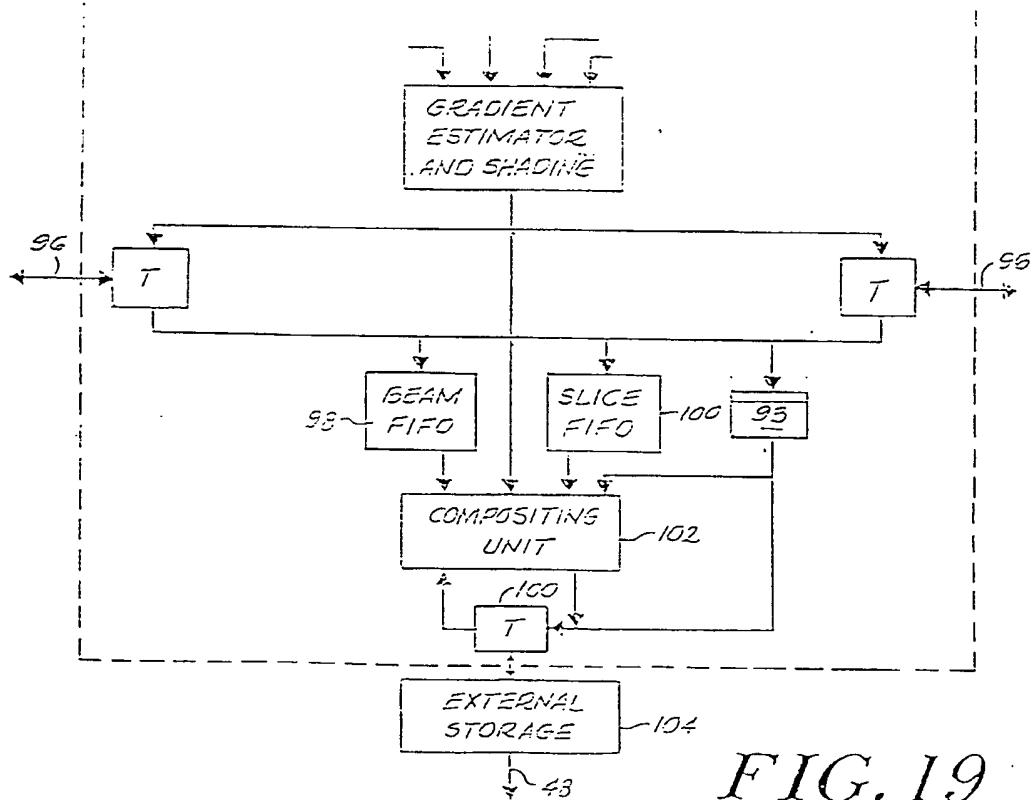


FIG. 19



European Patent
Office

EUROPEAN SEARCH REPORT

Application Number

EP 98 10 7592

DOCUMENTS CONSIDERED TO BE RELEVANT			CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	
A	<p>PFISTER H ET AL: "CUBE-4 - A SCALABLE ARCHITECTURE FOR REAL-TIME VOLUME RENDERING" PROCEEDINGS OF THE 1996 SYMPOSIUM ON VOLUME VISUALIZATION, SAN FRANCISCO, OCT. 28 - 29, 1996, 28 October 1996, pages 47-54, XP000724429 ASSOCIATION FOR COMPUTING MACHINERY * the whole document *</p> <p>---</p>	1-17	G06T15/00
A	<p>ANIDO M L ET AL: "THE ARCHITECTURE OF RIG: A RISC FOR IMAGE GENERATION IN A MULTI-MICROPROCESSOR ENVIRONMENT" MICROPROCESSING AND MICROPROGRAMMING, vol. 24, no. 1 - 05, 1 August 1988, pages 581-588, XP000105698 * page 584, left-hand column, line 26 - line 31 *</p> <p>---</p>	1,4	
A	<p>PHILLIPS D: "THE Z80000 MICROPROCESSOR" IEEE MICRO, vol. 5, no. 6, 1 December 1985, pages 23-36, XP000211949 * page 23, left-hand column, line 14 - line 24 * * page 27, left-hand column, line 5 - line 15 *</p> <p>-----</p>	1,4	<p>TECHNICAL FIELDS SEARCHED (Int.Cl.6)</p> <p>G06T</p>
<p>The present search report has been drawn up for all claims</p>			
Place of search	Date of completion of the search	Examiner	
BERLIN	5 November 1998	Burgaud, C	
CATEGORY OF CITED DOCUMENTS		T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document	
X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document			

THIS PAGE BLANK (USPTO)

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- BLACK BORDERS**
- IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- FADED TEXT OR DRAWING**
- BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- SKEWED/SLANTED IMAGES**
- COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- GRAY SCALE DOCUMENTS**
- LINES OR MARKS ON ORIGINAL DOCUMENT**
- REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.

THIS PAGE BLANK (USPTO)